Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Niklas Smal

# Real-time Global Photon Mapping:

## Adapting Global Illumination to Dynamic Environments in Real-time

Master's Thesis
Espoo, June 30, 2019

Supervisor: Jaakko Lehtinen, D.Sc. (Tech.), Professor, Aalto University School of Science
Advisor: Antti Hirvonen M.Sc. (Tech.)

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

**Author:** Niklas Smal

**Title:**
Real-time Global Photon Mapping: Adapting Global Illumination to Dynamic Environments in Real-time

**Date:** June 30, 2019 **Pages:** 101

**Major:** Computer Science **Code:** SCI3042

**Supervisor:** Jaakko Lehtinen, D.Sc. (Tech.)
**Advisor:** Antti Hirvonen M.Sc. (Tech.)

The focus of this thesis is to provide better methods to simulate the behaviour of light in synthesis of photo-realistic images for real-time applications. Improvements introduced in this work are related to indirect component of the illumination, also known as global illumination, in which the contributed light has already been reflected from surface at least once.

While there are a number of effective global illumination techniques based on precomputation that work well with static scenes, including global illumination for scenes with dynamic lighting and dynamic geometry remains a challenging problem. In this thesis, we describe a real-time global illumination algorithm based on photon mapping that evaluates several bounces of indirect lighting without any precomputed data in scenes with both dynamic lighting and fully dynamic geometry.

To make photon mapping possible within the performance limitations of the real-time rendering, we utilize and expand on several optimization methods, such as reflective shadow maps, stratified sampling and Russian Roulette. Furthermore, we introduce an improved distribution kernel for the screen space irradiance estimation of the photon mapping. Finally, we present a new filtering solution for photon mapping.

**Keywords:** photon mapping, global illumination, reflective shadow maps, a-trous wavelet filter

**Language:** English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

| | | | |
|---|---|---|---|
| **Tekijä:** | Niklas Smal | | |
| **Työn nimi:** | Fotonien kartoitus reaaliajassa: Epäsuoran valaistuksen soveltamista dynaamisille ympäristöille reaaliajassa | | |
| **Päiväys:** | 30. kesäkuuta 2019 | **Sivumäärä:** | 101 |
| **Pääaine:** | Tietotekniikka | **Koodi:** | SCI3042 |
| **Valvoja:** | Professori Jaakko Lehtinen | | |
| **Ohjaaja:** | Diplomi-insinööri Antti Hirvonen | | |

Opinnäytetyön painopisteenä on tarjota parempia menetelmiä valon käyttäytymisen simuloimiseksi reaaliaikaisten sovelluksien realistisessa kuvasynteesissä. Tässä työssä esitetyt parannukset liittyvät valaistuksen epäsuoraan komponenttiin, (tunnetaan myös globaalina valaistuksena), jossa valo on kulkenut ainakin yhden pintaheijastuksen kautta.

On olemassa tehokkaita globaaleja valaistustekniikoita, jotka perustuvat ennakkotietoon. Nämä tekniikat toimivat hyvin staattisten ympäristöjen kanssa, mutta dynaamisen valaistusta ja geometriaa ympäristöt ovat edelleen haastava ongelma. Tässä opinnäytetyössä kuvataan reaaliaikainen globaali valaistusalgoritmi, joka perustuu fotonikartoitukseen ja jossa arvioidaan useita epäsuoran valaistuksen askelmia ilman ennalta laskettua.

Jotta fotonikartoitus olisi mahdollista reaaliaikaisen renderoinnin suorituskyvyn määrittämissä rajoitteissa, käytämme useita optimointimenetelmiä, kuten heijastavia varjo-karttoja, kerrostettuja näytteitä ja venäläistä rulettia. Lisäksi esitämme parannetun distribuutiokernelin fotonikartoituksen säteilytysvoimakkuuden estimoinnille. Lopuksi esitämme uuden suodatusratkaisun fotonikartoitukseen.

| | |
|---|---|
| **Asiasanat:** | fotoni kartoitus, globaali valaistus, heijastavat varjo-kartat |
| **Kieli:** | Englanti |

# Acknowledgements

I would like to thank my employer, UL Benchmarks, for this opportunity and especially everyone from our 3DMark team. I also would like to thank professor Jaakko Lehtinen for supervising this work; Jani Joki and Antti Hirvonen for their feedback; and especially Max Aizenshtein for co-authoring the related Ray Tracing Gems article and overall answering all of my random questions.

Espoo, June 30, 2019

Niklas Smal

# Contents

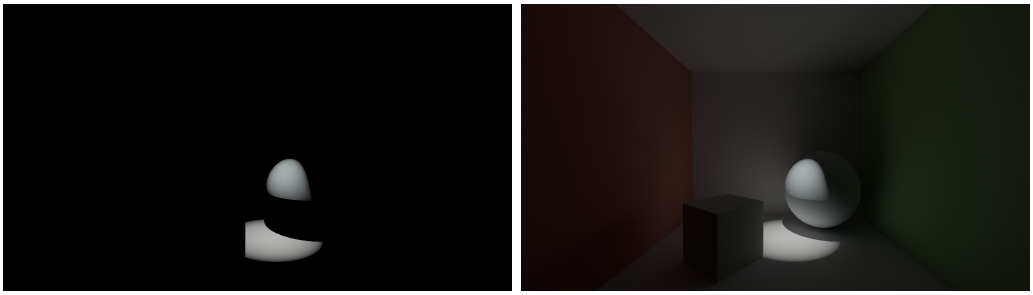
# Chapter 1

# Introduction

Physically based rendering can be defined as a process of using computers to synthesize photo-realistic images. This has multiple use cases ranging from architectural visualizations to production rendering in the film industry. However, in this thesis is focused mostly on *real-time rendering* applications, in which the image synthesis is done just prior to displaying the image. Therefore, the work required to generate a single image, commonly referred to as a *frame*, has a strict time restriction since the viewer must still perceive the result as a continuous sequence of images. The most common use case for this is in modern video games, where the dynamic nature of the gameplay requires frames to be rendered in real-time.

In order to pursue photo-realistic image quality, the behavior of light must be simulated as it travels in the virtual environment and is captured by some virtual camera. The most prevalent component of this is the interaction of light with surface materials by the ways of reflection, transmittance or absorption. Thus, to render an image, we must compute the amount of light the viewer receives from visible surfaces. This is referred to as computing *illumination* of these surfaces.

Computing illumination for cases in which light is applied directly from a light source to the surface, referred to as *local illumination*, is relatively simple: by knowing the position of the viewer, the light source and the reflecting surface, it is trivial to determine the path light travels. Unfortunately, this is not a realistic representation of the light's behavior in the real world as in addition to being reflected towards the viewer, light is also reflected back to the environment and thus contributing to the illumination of other surfaces. As a result, the number of paths light can take from the light source to the viewer becomes arbitrary. This type of indirect illumination is called *global illumination*. Contribution of the global illumination to the rendering quality has been shown in Figure 1.1. Regrettably, due to the aforementioned arbi-



*(a) Local illumination.* *(b) Local and global illumination.*

*Figure 1.1: Contribution of global illumination: Same simple scene with only local illumination (Figure 1.1a) and then with both local and global illumination. Without the contribution of global illumination, the illumination appears dark and unrealistic as surface not directly visible from the light source are completely dark.*

trary number of light paths contributing to the global illumination, achieving high quality global illumination comes with significant computational cost. Therefore, it is clear why computation of global illumination has remained a challenge especially in real-time rendering.

## 1.1 Motivation

This far we have demonstrated both the importance and the cost of global illumination. So how is this paradox solved in modern real-time applications? The most common solution is to precompute the global illumination, store the result to a data structure and then sample that data structure during the real-time rendering. This allows real-time access to high quality illumination, which is generally computed using similar *Monte Carlo rendering methods* to those that are used in movie production [10]. These rendering methods are explored in Chapter 2. Data structures for storing the precomputed illumination can vary greatly, but most of solutions are based on one of two approaches: a surface based data structure called *light maps* [20], that represent surfaces as unique samples in an image or a volume-based approach referred to as *illumination probes* [9] [27].

However, as the realm of possibility within real-time graphics grows with advancing capabilities of graphics hardware, one of the key factors has been an ever increasing dynamism of the rendered scenes. This is problematic for global illumination as the aforementioned precomputed solutions are by their core nature static, which leads to results of varying quality: in some cases relying on static global illumination data does not cause noticeable disconti-

nuities between static indirect lighting and per-frame calculated direct light, but in others a change in the indirect lighting would be highly desirable to improve visual quality.

Due to some the recent developments related to *ray tracing* (Chapter 2.5) in both graphics hardware and APIs (Chapter 2.5.1), the possibility of more dynamic global illumination solutions has expanded from the realm of academic research to the production of real-time graphics applications, such as video games. Unfortunately, this creates its own set of limitations: due to initially limited consumer adaption of the technologies required for these solutions, it is not viable approach to replace the application's entire rendering pipeline to accommodate the needs of the global illumination solutions, such as was done with the aforementioned film renderers. Therefore, this steered us instead of replacing the aforementioned static solutions to improve their quality when most beneficial, or i.e. wherever we can get "most bang for the buck" in terms of visual quality.

There has been several interesting approaches to add dynamism to precomputed illumination solution for both light maps [7] and probes [37] [24]. However, these solutions are still tied to their respective data structures. In comparison, our approach was designed to be completely separated from any existing solution or their data structures and thus can be applied regardless of the existing solutions.

Furthermore, since the dynamic global illumination technique is to be applied as an additional rendering feature when possible, it is essential that its adaption for the current rendering pipeline requires as little effort as possible: First, it must also require minimum artistic changes to set up. Second, it must be easily scalable for different time budgets depending on the target hardware. As a result, we set the following criteria for our technique:

- Compatible with precomputed global illumination solutions
- Does not require any additional data (e.g., unique UV coordinates or probe structures)
- Provides result with similar quality compared to current static techniques
- Easily scalable as a trade-off between quality and computation time
- No significant artist work required

## 1.2 Our Approach, Previous Work and Our Contributions

Based on the criteria described in the previous section, we opted to base our work on well-established rendering method, *photon mapping* [21]. It computes the indirect illumination in two separated passes: First, photon particles representing the light are emitted from a light source and traced through several surface reflections in the scene. Data related to these reflections is stored in a data structure referred to as a *photon map*. Second, we use this photon map to calculate global illumination by distributing the illumination of each photon to their surface neighborhood.

With photon mapping it is possible to compute global illumination with comparable quality to that sampled for precomputed illumination. In fact, photon mapping is one of the plausible algorithms used in the precomputation. Furthermore, the illumination being related to a light source from which the photons are cast allows us to define a sub-set of light sources for which the dynamic illumination would provide the highest increases in visual quality. Photon mapping is also well-suited to provide some rendering features that have been previously completely missing from real-time applications, such as realistic caustics and more extensive simulation of transmittance for transparent objects.

Unfortunately, achieving this using a naive implementation of photon mapping is not viable in the time budget of a frame. To solve this, we choose an approach with similar overall structure (Figure 1.2) to the previous work of McGuire and Luebke [26] while introducing several optimization methods (Chapters 4.2, 4.4 and 5).

In addition, we present a more optimal the definition of distribution kernel used for the aforementioned illumination calculation (Chapter 6.1) and provide comparison between to main approaches this computation can be done — *gathering* (Chapter 6.3) and *scattering* (Chapter 6.2).

Finally, we introduce a new filtering solution, which is well-suited for the low-frequency noise present in unfiltered photon mapping result (Chapter 7). This solution is an adaptation of previous filtering approaches by Dammertz et al. [15] and Schied et al. [33]

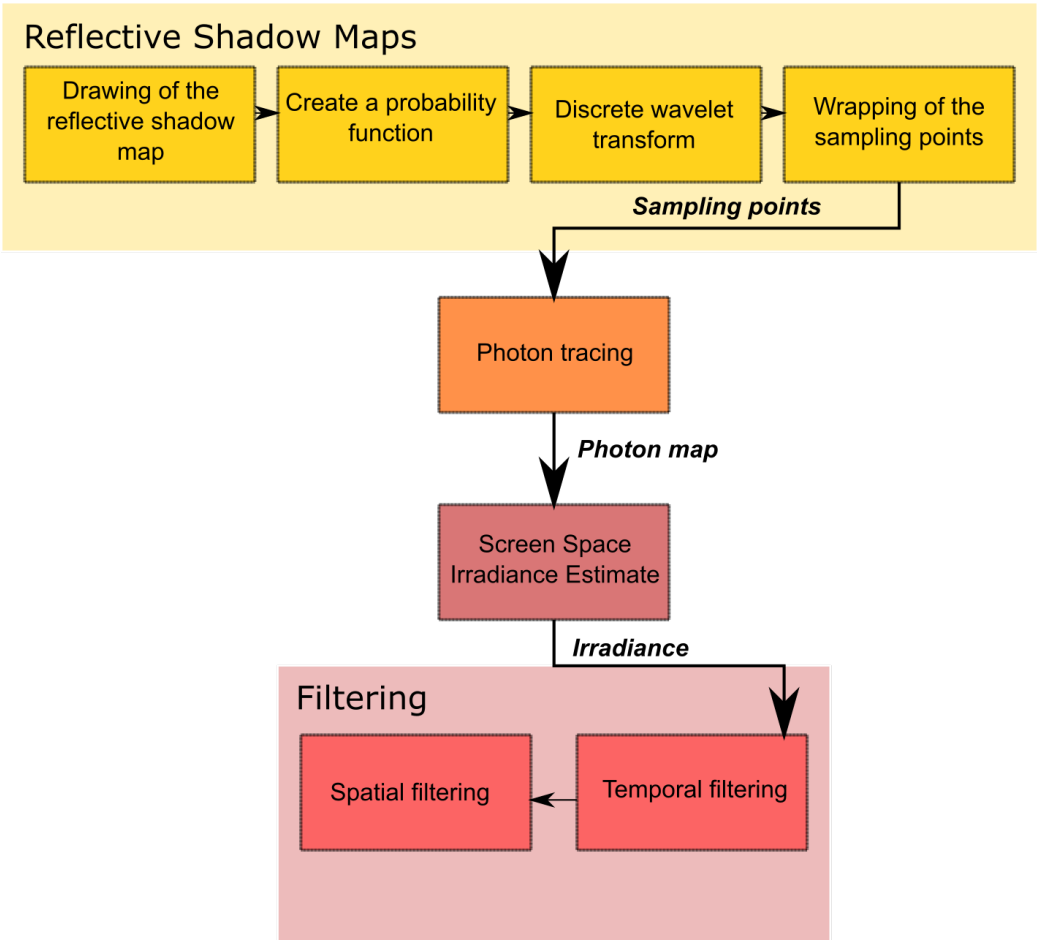



*Figure 1.2: Overview of algorithm different passes. As discussed above, we have the two main passes of photon mapping: creating of a photon map during photon tracing (Chapter 4) and then use that photon map to compute illumination by the pass called screen space irradiance estimate (Chapter 6). Tracing is optimized using reflective shadow maps (Chapter 5) instead of tracing for local illumination. Furthermore, illumination result is filtered using two filtering pass described in Chapter 7.*

## 1.3 Structure of the Thesis

The rest of the thesis is organized as follows:

In Chapter 2 we go over the background knowledge required for modern rendering process, such as the composition of a rendering scene and key features of modern rendering pipelines. Furthermore, we define the principal rendering concepts that our work relies on.

After the basics, in Chapter 3 we explore the rendering process in more detail: We introduce how to compute surface illumination by solving the light transport equation using a Monte Carlo integrator. We also delve deeper into how light interacts with a surface and how this is modeled using bidirectional reflectance distribution function.

In Chapter 4, we describe how to simulate indirect lighting by using photon tracing for both transparent and opaque surfaces. In addition, we explain how to utilize stratified sampling in relation to photon mapping. We also present an approach to speed up the photon tracing process using Russian Roulette optimization scheme for microfacet reflectance model. Finally, we discuss some of the details related to implementing photo tracing in modern rendering pipeline.

In Chapter 5, we discuss the optimization of the photon tracing by using reflective shadow maps to represent local illumination. Furthermore, we describe several methods, such as using importance sampling and Markov Chains, that can improve the sampling of the reflective shadow maps.

Next in Chapter 6, we go over how to generate screen space irradiance from a photon map: We explain how to minimize error during this process by selecting suitable distribution kernels. Following that, we introduce and compare two approaches to use these kernels to compute a screen space irradiance estimation.

In Chapter 7 we present an efficient filter solution that can compensate requirement of low sampling rate set by performance limitations in real-time applications.

In Chapter 8 we present our results and analyze both rendering quality and performance of the algorithm.

Finally, in Chapter 9 we summarize our work and discuss of possibilities for future work.

# Chapter 2

# Basics of Modern Rendering

Rendering is the process of generating a two dimensional (2D) image from a three dimensional (3D) scene as seen by some virtual observe. This process is defined by using three main components: *objects* defining the environment in the scene, *lights* providing illumination and a *virtual camera*. In this chapter we discuss these components on general level to provide elementary understanding required to follow further chapters. In addition, we describe some of the most essential rendering concepts, such as rasterization and ray tracing. Finally, we introduce the principal components of the modern graphics APIs that are used for real-time rendering. This is done to allow fundamental understanding of our photon mapping implementation's structure as well as choices behind it.

## 2.1 Scene objects

First of all, we must define environment in the virtual 3D world we wish to render. This is achieved by a set of 3D objects consisting of *geometry* [31, Ch 3.6] and *material parameters* [31, Ch 9] .

**Geometry** Geometry defines the shape of an object's surface, for which the most common approach is by using a *mesh*. A mesh is composed of a set of triangles, which is logical as they are the simplest forms of the basic geometries in 3D space. These triangles define the tangent plane of the surface by using three *vertices*, which include their positions as a *vertex attribute*. Vertices are linked together as a triangle using a list of *indices*. Furthermore, vertices can also have other vertex attributes, such as a *vertex normal* (a direction vector perpendicular to the surface) and coordinates for texture sampling. These attributes are then interpolated between vertices to

specify attributes for the surface points at their tangent plane.

**Materials** Materials describe the properties of the surface. They can be defined either as a constant, a vertex attribute or given as textures which are then sampled using interpolated texture coordinates. These properties can vary depending on the shading model used in illumination calculation for the surface, but most of the modern physically based shading models include *albedo*, *normal*, *metalness* and *roughness* property parameters. Use of these properties in shading is explained later in Chapter 3.1.

## 2.2 Camera

The intention of the virtual camera [31, Ch 6] is to deduce how to project a 3D scene to a 2D image for display. The most common solution is to use a perspective projection to project objects that are inside camera's *view frustum* and located between two clipping planes (one closer to camera called *near plane* and the other *far plane*) into the virtual film plane. This plane is usually referred as *screen space* and consist of a number of *pixels* that represent discrete samples of the final image.

Before applying the projection matrix we must first transform the rendered objects to *view space*, which is defined as having the camera position as its origin and its XY-plane being parallel to near and far planes of the view frustum. Once the scene is transformed to view space, we can apply the projection. This results coordinates being transformed to *normalized device space* (NDC). This is demonstrated in Figure 2.1.

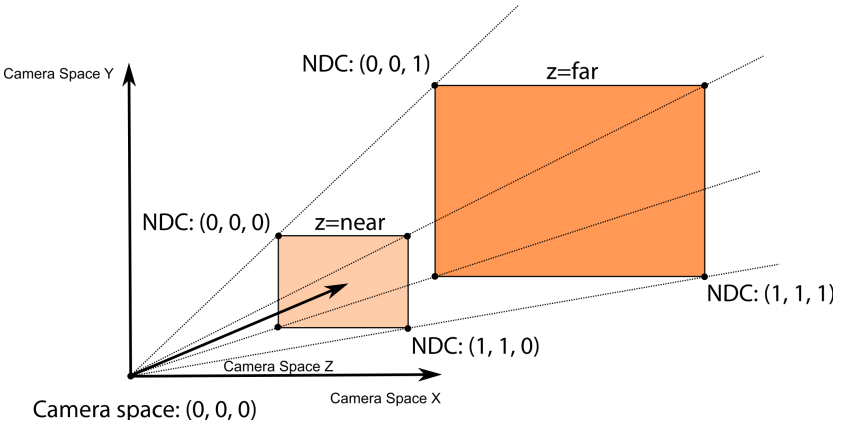



*Figure 2.1: Relations between NDC and view space of the camera. Figure inspired by [31, Ch 6.2].*

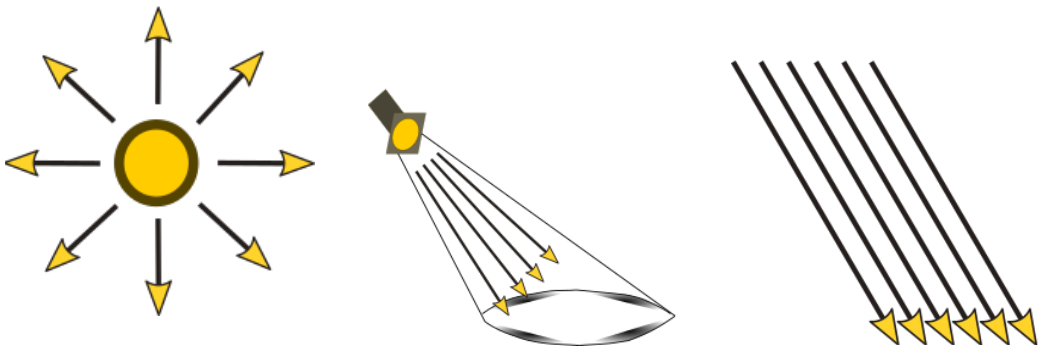
## 2.3 Lights

In real world "lights" can be defined as surfaces emitting light and thus illuminating their environment. This is as also a valid approach in computer graphics and these light sources are commonly referred to as *area lights*. However, given the limited time budget for computation in real-time rendering, it is common to rely on approximations that can be solved analytically. These approximations are referred to as *virtual lights sources* and they have three prevalent types — *point*, *spot* and *directional* [6, Ch 5.2]. These light types are showcased in Figure 2.2.

**Point lights** are the most simplest of the light types. They emit light from a single position to all directions.

**Spot lights** are similar to the real-world use case a flashlight: they project a tight beam of light from a point to a defined direction as a cone or a frustum.

**Directional** light is commonly used to represent sun light as it is applied to all surfaces in the scene from the same direction.



*(a) point light* *(b) Spot light* *(c) Directional light*

*Figure 2.2: Three most common virtual light types.*

### 2.3.1 Shadow maps

To compute any realistic surface illumination we must determine the visibility of the contributing lights from that surface, i.e. we must define if the surface

is in a shadow. In real-time rendering this is most commonly done using *shadow maps* [6, Ch 7.2]. Shadow maps are generated by rendering the scene from the light source and storing the distance to the closest surface. For spot lights this means using the same type of virtual camera that we use for main rendering pass while replacing the view frustum with frustum of the light. In the case of directional lights, the process is fairly similar but using a slightly different type of projection. Visibility is usually ignored for the points lights as they lack a well-defined direction and thus would require much more computation work to render a shadow map.

## 2.4 Graphics Pipeline using Rasterization

With access to geometry and virtual camera, we have all the building blocks required to render a 3D scene. But how use these blocks to generate a 2D image in the most optimal way possible? The most prevalent algorithmic approach to resolve screen space pixel values from surfaces within NDC space by is using rasterization. This is also the default approach used by rendering pipelines of every modern real-time graphics API. These pipelines are furtheron referred as *graphics pipelines* [6, Ch 2].

**Rasterization** [6, Ch 2.4] is an algorithm that solves the *visibility problem* (i.e., defining which of the surfaces is seen for a pixel) by projecting the triangle to the screen space and keeping the value that is closest to the camera. In order to do this, it must store the NDC depth value of the closest sample in addition to the sample value itself. These depth values are known as *depth buffer* or *depth stencil* [6, Ch 2.5.2].

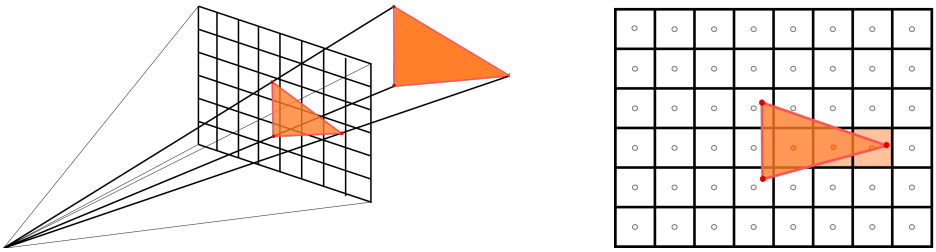


*Figure 2.3: Illustration of rasterization. First, triangle is projected from NDC to screens space. Second, pixels are sampled to define if they are within the triangle. These samples are also compared to the depth buffer and samples further away are rejected.*

### 2.4.1 Basics of Modern Graphics Pipeline

As mentioned above, all of the modern rendering APIs rely on rasterization as their primary method of drawing images. Next we are present a quick introduction into the structure of these pipelines to provide some understanding their usage later on. However, this introduction is very superficial since it mostly falls out of the scope of this work.

Generally speaking the work for graphics pipeline is defined on central processing unit (CPU) by recording *draw calls* [3] for scene objects we intend to draw. These draw calls are then bind to the objects' geometry and material data required for the drawing. Draw calls initiate a number of *shader invocations*, which can be thought of as instances of the pipeline's programmable components known as *shader programs* [6, Ch 3.8]. For the sake of clarity we ignore everything but the two main shader programs, which are later used for our scattering implementation in Chapter 6.2 — *vertex* and *pixel shaders*.

**Vertex shaders** [6, Ch 3.5] are invoked for the three corners of the drawn triangles and contain the transformation of these vertices' positions to NDC. Furthermore, we define all vertex attributes that we wish to be interpolated and transferred to pixel shaders to be used in surface shading.

**Pixel shaders** [6, Ch 3.8] are a type of shader that is invoked for all pixels resulted from triangle's rasterization. Therefore, these pixels are closer to the camera than current depth buffer and thus the pixel shader is used to compute surface's output value for the pixel in question. Furthermore, it is not mandatory for pixel shader outputs to replace values behind its depth stencil since the rasterization pipeline has several modes that allow to modify its behavior: e.g., in Chapter 6.2 we use *blending* [6, Ch 5.5] for summation of the pixel shader outputs for overlapping surfaces.

There is also a third major shader type — *compute shader* [6, Ch 3.10]. These are general purpose shaders, that instead of can be programmed for computation work outside the graphics pipeline while using an arbitrary number of shader invocations. In addition, one of the key advantages of compute shaders is access to *shared memory* [1], which allows using a more performance efficient memory cache accessible to a set of invocations. We utilize this feature e.g. in our filtering implementation (Chapter 7).

Finally, we mentioned above that the most common case for graphics pipeline work is to be defined by the CPU. However, this is not always possible as data required to define invocation parameters might be generated by the preceding passes. This data could be copied back to the CPU, but this

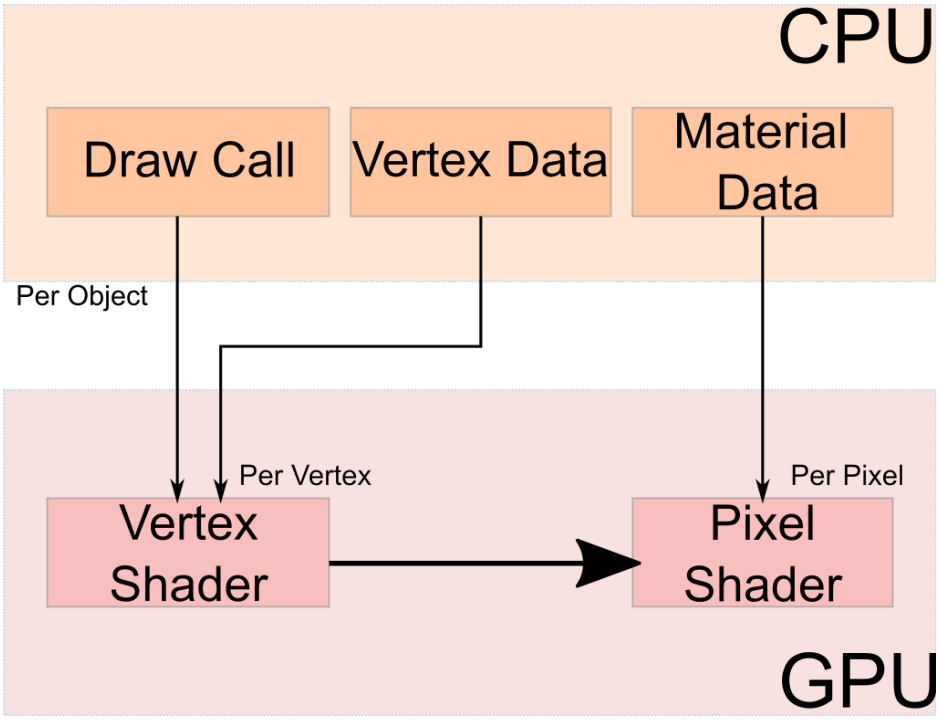



*Figure 2.4: Visualization of different interactions in the rendering pipeline. Please note that this is an extreme simplification of the rendering pipeline and for more complete description please refer to e.g. Chapter 2 of Real-time Rendering book [6].*

would have significant performance implications. Therefore, graphics APIs allow the possibility for shader invocations to be also defined by a modifiable data structure residing in the memory of graphics processing unit (GPU). These are called *indirect invocations* [4]. We utilize indirect invocations for both of our radiance estimation approaches in Chapter 6.

## 2.5 Ray Tracing

Ray tracing is the most essential concept behind our algorithm. However, in literature "ray tracing" can have multiple meanings. In some contexts it is referred to as an alternative rendering algorithm to rasterization that solves the visibility problem by tracing a ray from a camera though each screen space pixel while evaluating the ray's triangle intersections to find the closest surface. However, in this thesis we call this rendering method "ray casting" and ray tracing is specified in more generalized terms [31, Ch 2.5]: ray tracing is defined only as casting a ray from point $O$ to direction direction $\widehat{\mathbf{d}}$ to find the hit point $P$ at the ray length $t$. Thus, ray can formally defined

as

$$P(t) = O + \widehat{\mathbf{d}}t \tag{2.1}$$

Sometimes it is enough to find any hit point within a limited ray length (e.g., when using ray tracing to define shadows) but in most cases $P(t)$ is located in the closest surface along the ray and this can be assumed unless mentioned otherwise.

So why is ray tracing so essential for our work? It is because ray tracing allows efficient computation of visibility function queries by solving the Equation 2.1 for a single ray. To do this utilizing rasterization, for each query we would have to render an entire image assuming that rays are incoherent, i.e. if there is difference in $O$ or $\widehat{\mathbf{d}}$ is outside of the previous view frustum. This would obviously lead to unfeasible amount of extra work. In contrast, rasterization can be a viable solution for coherent set of rays, such as shown in Chapter 5.

**Ray differentials** [19] Ray differentials expand the concept of rays to *ray cones* [10, Ch 5.1.3] as a representation of "neighboring" rays with slightly different origins and directions. I.e., ray differentials is a way to express slightly varying rays as they are reflected at a surface without the need of examining each individual ray. Formally, the ray differential can be defined as:

$$\left\{ \frac{\partial O}{\partial x}, \frac{\partial O}{\partial y}, \frac{\partial \hat{\mathbf{d}}}{\partial x}, \frac{\partial \hat{\mathbf{d}}}{\partial y} \right\}, \tag{2.2}$$

where $x$ and $y$ are screen space pixel coordinates. It is worth noting that this does not necessarily have to be screen space of the virtual camera: our distribution kernel approximates ray differentials for photons within the frustum of a spot light (Chapter 6.1). Furthermore, the concept of ray differentials in surface reflection is visualized in Figure 2.5.

Given the amount of geometry in modern rendering scenes, computing a naive intersection between its triangles and a ray is not feasible even in offline rendering, not to mention in real-time. Therefore, an *acceleration structure* is used to hasten the process. However, acceleration structures, and making ray tracing efficient in general, is an extensive field of study and beyond scope for this thesis. Therefore, ray tracing is considered a black box solution that is implemented using API described in the following section.

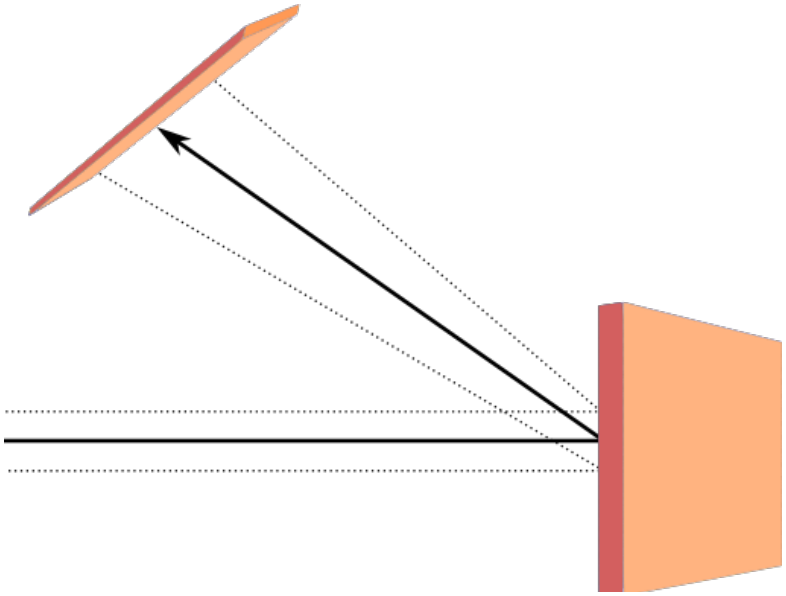


*Figure 2.5: Illustration of the ray cone: ray and its extended ray cone (dashed). The spread angle is defined by microsurface normal variance, i.e. roughness of the surface (see Chapter 3.1), and the normal slope of the surface. Figure inspired by [10].*

### 2.5.1 Real-time Ray Tracing and DXR

In the fall of 2018, Microsoft introduced DirectX Ray Tracing (DXR) as a part of its commonly used graphics API, DirectX. This is a significant step forward as it allows access to an efficient ray tracing pipeline from DirectX. Furthermore, it enables some improvements that allow ray tracing to be efficient enough to be viable in real-time applications, such as utilizing ray tracing specific hardware in the graphics card.

In this work we utilized DXR as a black box solution for the ray tracing process. Details of the implementation are out of the scope for the thesis, but for the context of understanding provided coding samples, we explain two shader types used in DXR:

**Ray Generation shaders** are an initial pass that allows us to call the ray tracing pipeline. These shaders are invoked in a similar manner to compute shaders. In addition to the ray, we define a *payload* data structure that allows us to transfer data between passes.

**Closest Hit shader** is invoked for each successful ray tracing operation. If there is no hit point, a *miss shader* is invoked. In the closest hit shader we have access to the ray length and other parameters necessary to define the material properties of the surface.

# Chapter 3

# The Light Transport Equation

In the introduction we provided some intuition related to behavior of the light as it traversals in the scene. However, we must formalize this behavior in order to compute the illumination.

The light transport equation (LTE) [31, Ch 14.4] is the governing equation that describes the equilibrium distribution of radiance in a scene. It gives the total reflected radiance at a point on a surface in terms of emission from the surface, its reflection distribution function and the distribution of incident illumination arriving to the point. During the rendering, we compute the illumination in screen space by solving this equation for surfaces visible from the virtual camera. However, evaluating LTE is difficult because it is affected by geometrical and material properties of all surfaces in the scene. Algorithms taking into account this complexity are usually referred as *global illumination* and those that do not as *local illumination*.

**Radiometric Quantities** [31, Ch 5.4] This far we have described light's contribution to the scene only as illumination, but before exploring this further we must expand the definitions of this contribution. First, the power of light, $\phi$, is described as *radiant power* or *radiant flux* and it is measured as joules per second ($J/s$). Second, the amount of light emitted, reflected or received by the surface is represented by *irradiance* $E$, which is measured as watts per square meter ($W/m^2$). Third, expand the irradiance by defining it per solid angle resulting in *radiance* $L$. Unit of radiance is watts per square meters and steradian ($w/(m^2 sr)$).

Let us make an assumption that there is no participating media in the light's path and therefore radiance is constant along those paths. Thus, we can relate the incident radiance at the point to the outgoing radiance from another point:

$$L_{\mathrm{i}}(\mathrm{p},\omega) = L_{\mathrm{o}}(P(\mathrm{p},\omega),-\omega) \tag{3.1}$$

where $P$ is the ray function. This allows formulate LTE in a format commonly known as *rendering equation* [31, 14.4]:

$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_{H_i^2} f(p,\omega_o,\omega_i) L(p,\omega_i) \mid \cos\theta_i \mid \, \mathrm{d}\,\omega_i \tag{3.2}$$

where $p$ is reflective surface, $\omega_i$ the direction of the incoming light and $\omega_o$ the view direction of the camera. These are shown in Figure 3.1.

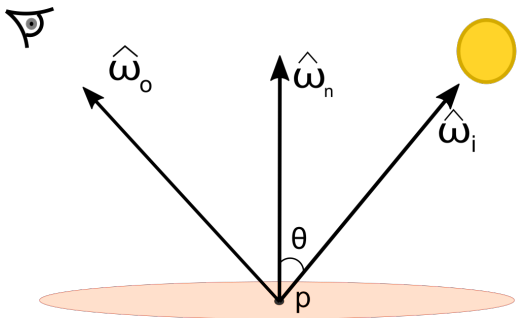



*Figure 3.1: Illustration of the directions used in Equation 3.2 with $\omega_n$ being the surface normal.*

Rendering equation results in outgoing radiance $L_o(p,\omega_o)$ from the surface $p$ and thus how much light the surface contributes to outgoing direction $\omega_o$. Assuming the outgoing direction points to the virtual camera through pixel of its screen space, this allows you the define pixel's value in the image we are rendering. Next, let us examine the different terms contributing to this result.

**Emissive term** $L_e$ represents the radiance emitted to the camera from surface $p$. Due to computational limitations, as discussed in Chapter 2.3, using geometry as a light source is an uncommon use case but not unheard of. Thus, surface luminance is usually only applied additively to its own illumination while not contributing to the lighting of its environment. Therefore, it is generally defined by material parameters of the surface and can be solved by simply sampling those parameters. Furthermore, most surfaces are not light-emitting which leaves the following integral as the key of solving the rendering equation.

**Integral** $\int_{H_i^2} f(p, \omega_o, \omega_i) L(p, \omega_i) \mid \cos\theta_i \mid \, \mathrm{d}\,\omega_i$ represents the quantity of light reflected by the surface based on the amount of light it receives from its environment. This is defined by integrating the incoming radiance over hemispherical directions, $H$, while taking to account matching *bidirectional reflectance distribution function* (BRDF) [29] and *cosine term*.

BRDF models surfaces behavior to reflect light and it is therefore determined by surface's material parameters. This is vital in order to achieve photo-realistic rendering as reflective properties surface materials can vary greatly. Cosine term represents weakening of the outward irradiance due incident angle and it can be derived from the dot product of $\omega_i$ and $\omega_n$. Both BRDF and cosine term are derived in the Chapter 3.1.

*Incoming radiance*, $L(p, \omega_i)$, can be defined as either coming directly from a light source as local illumination or being reflected from other surfaces as global illumination. Local illumination can be solved using analytical solutions for all virtual light types. Unfortunately, accounting for indirect light is more complicated: We must solve incident radiance at $p$ from all visible surfaces or i.e., to accumulate incoming radiance from all the hemispherical directions. Unfortunately, the only way to determinate the incident radiance from a surface is to determine its outgoing radiance (Equation 3.1) by solving the rendering equation at that surface. This obviously becomes a recursive process. To handle this, one approach is to derive more flexible form of LTE known as *path integral formulation of light transport* [41][31, Ch 14.4.4].

Finally, we must be able to solve the integral. This is usually done by using computational methods, most commonly *Monte Carlo integration* [31, Ch 13]. As a result, these rendering algorithms are referred to as *Monte Carlo rendering algorithms*.

## 3.1 Bidirectional reflectance distribution function

When light hits a surface, it scatters and a part of it is reflected back to the environment. This interaction is in the represented by bidirectional reflectance distribution function, $f(p, \omega_o, \omega_i)$. In layman terms this function can be said to be the ratio of incoming irradiance is being reflected to $\omega_o$. The formalized definition is as follows [31, 5.6.1]:

Let us consider $\omega_i$ as a differential cone of directions, the different irradiance is defined as:

$$\mathrm{d}E(p,\omega_i) = L(p,\omega_i)\cos\theta_i \,\mathrm{d}\,\omega_i \tag{3.3}$$

Due to this irradiance, a differential amount of radiance is reflected to $\omega_o$. Because of linearity assumption from geometric optics this differential irradiance is proportional to the irradiance and this constant proportionality defines the BRDF for surface $p$:

$$\mathrm{d}L_o\left(p,\omega_o\right) \propto \mathrm{d}E\left(p,\omega_i\right)$$

$$f\left(p,\omega_o,\omega_i\right) = \frac{\mathrm{d}L_\mathrm{o}\left(p,\omega_o\right)}{\mathrm{d}E\left(p,\omega_i\right)} = \frac{\mathrm{d}L_o\left(p,\omega_o\right)}{L\left(p,\omega_i\right)\cos\theta_i \mathrm{d}\omega_\mathrm{i}}$$

From this we can derive:

$$\mathrm{d}L_o(p,\omega_o) = f(p,\omega_o,\omega_i)L(p,\omega_i) \mid \cos\theta_i \mid \ \mathrm{d}\,\omega_i$$

This shows the relation of incoming radiance, BRDF, and the cosine term in the rendering equation, which can be derived by adding the integration over $H$ to hemispherically accumulate $\mathrm{d}L_o(p,\omega_o)$.

Due to conservation of energy, the amount of light reflected must be less or equal to incident light:

$$\forall\omega_o, \int_{H_i^2} f\left(p,\omega_i,\omega_o\right)\cos\theta \mathrm{d}\omega_i \leq 1$$

### 3.1.1 Diffuse and Specular Reflectance

Modeling surface materials using BRDFs is an extensive field of research and here we only provide a simple overview of the BRDF functions used in our work — *Lambertian* and *Cook-Torrance* [12] BRDFs. Furthermore, we focus mainly on the background knowledge required for understanding the surfaces interactions of the photons in Chapter 4 and thus most of the details and evaluation between different models are beyond the scope of this work. For further information, please refer to works of Walter et al. [43], which provide an extensive coverage of the subject.

The most simplest BRDF is known as *Lambertian* BRDF in which surface reflects light evenly over the upper hemisphere. This effect is also called *diffuse reflection*. Formally, this can be defined as

$$f_{Lambert}(p,\omega_o,\omega_i) = \frac{c}{\pi}, \tag{3.4}$$

where $c \in [0, 1]$ and is the *albedo color* defined by surface's material parameters. Visualization of the physical light interaction represented by diffuse reflection is presented in Figure 3.2.

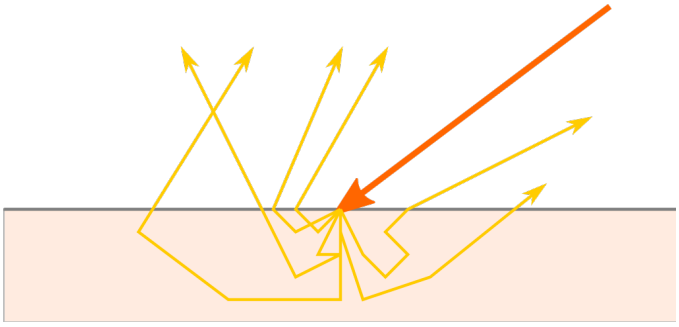


*Figure 3.2: Diffuse reflection: Incoming light enters the surface material and results in subsurface scattering where the light energy scatters within the material until it re-emerges from the surface, typically after undergoing partial absorption. This behavior is simplified as an uniform hemispherical distribution of the outgoing directions at the reflection point, since in most cases simulation of subsurface scattering is too computationally expensive. Figure inspired by [18].*

Nonetheless, using only Lambertian BRDF is simply insufficient for the requirements of modern rendering. In fact, Lambertian BRDF achieves poor results even for diffuse reflections and it is common even for real-time applications to rely on more complex models, such as *Oren-Nayar* [30]. However, this is not significant in the context of photon mapping and therefore for clarity we retain using Lambertian to model diffuse reflectance. More important for us is to expand the reflectance model by adding a *specular* component (Figure 3.3). This allows us the create substantially better representation of significant amount materials compared to relying only on a diffuse model.

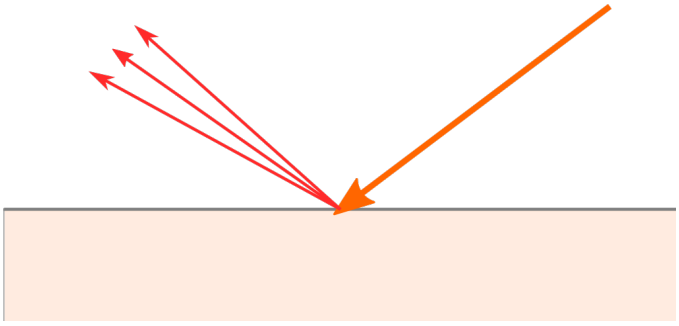


*Figure 3.3: Specular reflection: Light is reflected at the surface to near mirror-like directions. Figure inspired by [18].*

Specular reflection is modeled by using *Cook-Torrance BRDF*, which is a widely adapted physically based BRDF model in real-time rendering. Therefore, reflectance can be defined as:

$$f(p, \omega_o, \omega_i) = k f_{Lambertian} + (1 - k) f_{Cook-Torrance},$$

where $k \in [0, 1]$ is the ratio light that is diffusely reflected. This ratio is defined by surface parameter called *metalness* with metallic surfaces having more dominant specular component. It is worth noting that combining these BRDFs is not exactly physically based as most of real materials are combinations of layers with different surface properties and these layers are handled differently in different shading models. However, blending approach presented above is a commonly used approximation. This merging of the BRDFs can be also seen as combining multiple *reflection lobes* as shown in Figure 3.4.

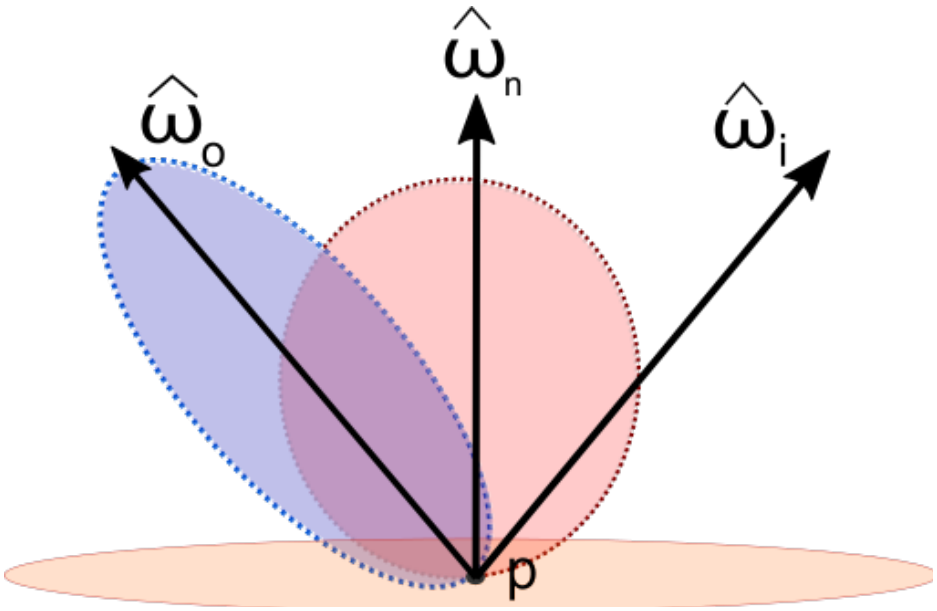



*Figure 3.4: Reflection lobes of specular (blue) and diffuse (red) reflections in 2D. The size and shape of these lobes in defined by the their BRDF functions. Multiple lobes can be combined to achieve better representation of the surface reflection.*

The specular component, Cook-Torrance BRDF, is based on *microfacet theory*, where a surface is considered as a collection of small planar mirrors that reflect perfectly. Formally, this is defined as

$$f_{CT} = \frac{DFG}{4(\omega_o \cdot \omega_n)(\omega_i \cdot \omega_n)}, \tag{3.5}$$

in which $D$ is the distribution function, $F$ is the *Fresnel term* and $G$ is the *bidirectional shadowing-masking function.*

*Microfacet normal distribution function* $D$ describes the statistical distribution of the microfacet surface normals $\omega_m$ over a macro surface. Concept of microfacet surfaces and normals is explained in Figure 3.5. As a distribution function we choose *Trowbridge-Reitz GGX distribution*:

$$\chi(x) = \begin{cases} 0 & \text{for} \quad x \leq 0 \\ 1 & \text{for} \quad x > 0 \end{cases}$$

$$D(m, n, \alpha) = \frac{\alpha^2 \chi(\omega_m \cdot \omega_n)}{\pi((\omega_m \cdot \omega_n)^2(\alpha^2 - 1) + 1)^2} \tag{3.6}$$

In which the $\alpha$ is defined by material parameter *roughness.*

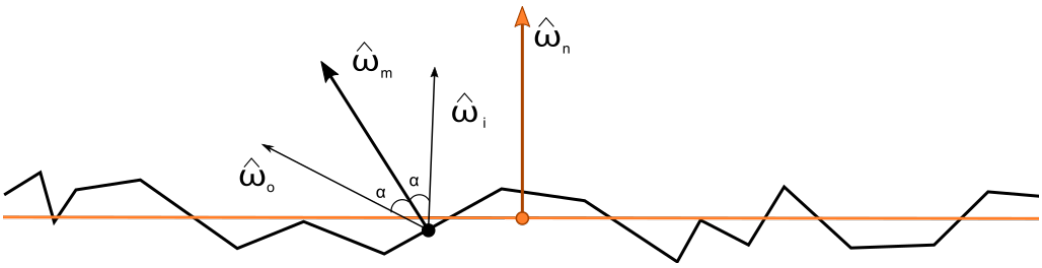



*Figure 3.5: The difference between micro and macro surfaces: We can see the macro surface of geometry as a simplification of much more complex micro surface structure. However, we can modify to scattering function to match micro surface and thus i.e. the difference can not be seen directly but it still affect the light distribution. Since micro surfaces are perfectly reflection mirrors, we can define the micro surface normal $\omega_m$ as a half vector of view and light directions being reflected at the micro surface. Figure inspired by [43].*

The bidirectional shadowing-masking function $G$ defines the amount of self-shadowing caused by the microfacets and describes the fraction of the microsurfaces for which both incoming and outgoing directions are visible. This is visualized in Figure 3.6. For this we use *Smith's* [39] shadowing function, formally:

$$G(\omega_i, \omega_o, \omega_m, \omega_n, \alpha) = G_p(\omega_o, \omega_m, \omega_n, \alpha) G_p(\omega_i, \omega_m, \omega_n, \alpha) \tag{3.7}$$

$$G_p(\omega, \omega_m, \omega_n, \alpha) = \chi(\frac{(\omega \cdot \omega_m)}{(\omega \cdot \omega_n)}) \frac{2}{1 + \sqrt{1 + \alpha^2 \frac{1 - (\omega \cdot \omega_m)^2}{(\omega \cdot \omega_m)^2}}}$$

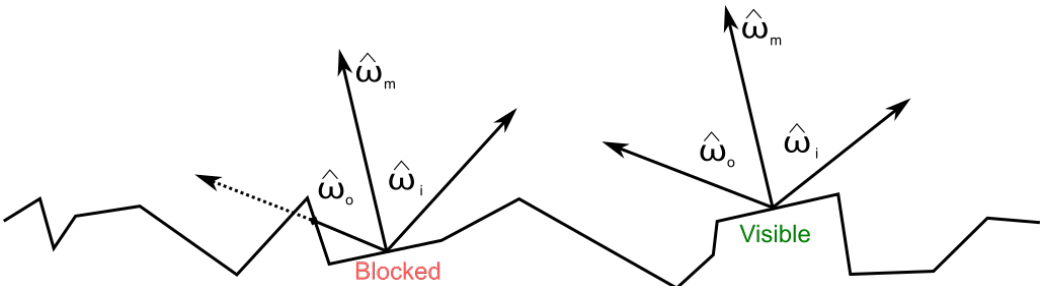



*Figure 3.6: Shadowing Masking Function: Blocked and visible fraction with two surfaces that have the same $\omega_m$. Figure inspired by [43].*

Finally, the Fresnel term simulates the lights interaction with surface in different angles. It is computed using Shlick approximation [34]:

$$F = F_0 + (1 - F_0)(1 - \cos(\theta))^5 \tag{3.8}$$

## 3.2 Monte Carlo Integrator

We have now defined all of the terms within the integral of the rendering equation (Equation 3.2) and thus to solve the equation, we are left with solving the integral itself. Intuition behind the integral is quite clear: in order to accumulate incoming light from the environment, we must determinate incident light from all hemispherical directions. Unfortunately, this integral equation do not generally have analytic solutions and therefore we must rely on numerical methods. Furthermore, high dimensionality and discontinuous nature of the integrals make standard numerical methods, such

as trapezoidal integration and Gaussian quadrature, poorly suited due low conversion rate. [31, Ch 13]

Monte Carlo integration is the most common solution for this problem as it is much better suited for the task. It uses sampling of multiple randomized arbitrary points within the integral's domain to estimate its values. Thankfully, in the case of rendering equation this is easy to implement since we can readily generate arbitrary samples. Let us then define a basic Monte Carlo estimator:

$$F_N = \frac{1}{N} \sum_{N}^{i=1} \frac{f(X_i)}{p(X_i)}$$

given random variables $X_i \in [0, 1[$ that are drawn from arbitrary *probability distribution function* (PDF) $p(x)$. This form of the definition takes into account the non-uniform random variables. This is essential since careful selection of the samples within PDF have a significant effect reducing variance of the estimator. However, this has the limitation that for $f(x) > 0$ the $p(x)$ must be non-zero. Furthermore, we can easily define *expected value* of this estimator as $\int_A f(x) \, \mathrm{d}\, x$:

$$\begin{aligned} E[F_N] &= E\left[\frac{1}{N} \sum_{N}^{i=1} \frac{f(X_i)}{p(X_i)}\right] \\ &= \frac{1}{N} \sum_{N}^{i=1} \int_A \frac{f(x)}{p(x)} p(x) \, \mathrm{d}\, x \\ &= \frac{1}{N} \sum_{N}^{i=1} \int_A f(x) \, \mathrm{d}\, x \\ &= \int_A f(x) \, \mathrm{d}\, x \end{aligned} \tag{3.9}$$

We can see that the expected value can be evaluated by averaging a large set of outcomes from the random variable. Using *weak law of large numbers* [5] we can conclude that it is highly probably to find a result that is statistically very likely to be close to the true answer and with error approaching zero as $N$ is approaching infinity:

$$\lim_{N->\infty} P(\mid \mu - E[x] \mid > \varepsilon) = 0$$

By using the Monte Carlo method we can derive rendering from Equation 3.2 into:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \frac{1}{N} \sum_{j=1}^{N} \frac{f(p, \omega_o, \omega_i) L(p, \omega_i) \mid \cos \theta_i \mid}{p(\omega_j)} \tag{3.10}$$

Bias for the estimator can be defined as

$$Err[I] = \frac{1}{N} \sum_{N}^{i=1} \frac{f(X_i)}{p(X_i)} - \int_A \frac{f(x)}{p(x)} p(x) \, \mathrm{d}\, x$$

and the estimator is unbiased if the expected value is equal to the correct answer, i.e. $Err[I] = 0$. Furthermore, variance for an unbiased Monte Carlo estimator [22, p. 36] is:

$$Var = O(\frac{1}{N})$$

and consequently *root mean squared error* ($RMSE$) is $O(\sqrt{N}^{-1})$. The proof for this is presented in Appendix A. Unfortunately, this brings up one of the main downsides of Monte Carlo methods: in order to half the error we must take four times as many samples. Therefore, we require more sophisticated approaches to reduce the error and thus increase the rendering quality.

**Importance Sampling** [31, 13.10] Since the Monte Carlo estimator never truly converses due to a finite amount of samples, we must consider ways to lower the error within the sampling budget available to us. One solution for this is to utilize importance sampling: integrator converges more quickly if the samples are taken from a distribution $p(x)$ that is similar to the function $f(x)$ in the integral. I.e., we concentrate samples to where the value of the integral is high and thus acquire an accurate estimate more efficiently.

## 3.3 Path Integral Formulation of Light Transport

Path integral formulation of light transport evaluates LTE as an integral over *paths* which are defined as points in a high dimensional *path space*. This is done to avoid unwieldy recursive behavior of solving global illumination by replacing the hemispherical integral with explicit integral over the path space: instead of the hemispherical accumulation, a path is defined only for a single direction of incoming light per surface reflection. Furthermore, these paths

can contain an arbitrary number of surface reflection, also referred as its *bounces*. This removes the nested integrals when evaluating several bounces of indirect illumination by determining path's lighting as accumulating the radiance of its bounces while accounting for previous BRDFs and cosine terms. Formally, this can be defined as:

$$P\left(\overline{\mathcal{P}}_n\right) = \int_A \int_A \cdots \int_{A_e(\mathcal{P}_n \to \mathrm{p}_{n-1})} \times \left( \prod_{i=1}^{n-1} f\left(\mathcal{P}_{i+1} \to \mathcal{P}_i \to \mathcal{P}_{i-1}\right) G\left(\mathcal{P}_{i+1} \leftrightarrow \mathcal{P}_i\right) \right) \mathrm{d}A\left(\mathcal{P}_2\right) \cdots \mathrm{d}A\left(\mathcal{P}_n\right)$$

In practical cases, and especially in real-time rendering, these paths have a limited amount of bounces. The perhaps most clear example of this is *path tracing*.

### 3.3.1 Path tracing

Path tracing is the most commonly used of the Monte Carlo rendering algorithms and in recent years it has become the industry standard for offline rendering [10]. We also use a path tracer as a comparison method for our work. Path tracing can be seen as an extension of ray casting and simplified description of the algorithm follows a straight forward loop:

1. Cast a ray from the previous surface or from the camera to find the current surface.
2. Cast a shadow ray to define the visibility of the light source(s).
3. Solve the rendering equation for the current surface.

To compute the radiance of a path over multiple bounces, we repeat these steps for each bounce while accumulating the total radiance as a sum of bounces' radiance. It is worth noting that each bounce's radiance is also affected by the BRDF of the previous bounces since its contribution to the camera is affected by reflections from those surfaces. This process is demonstrated in Figure 3.7.

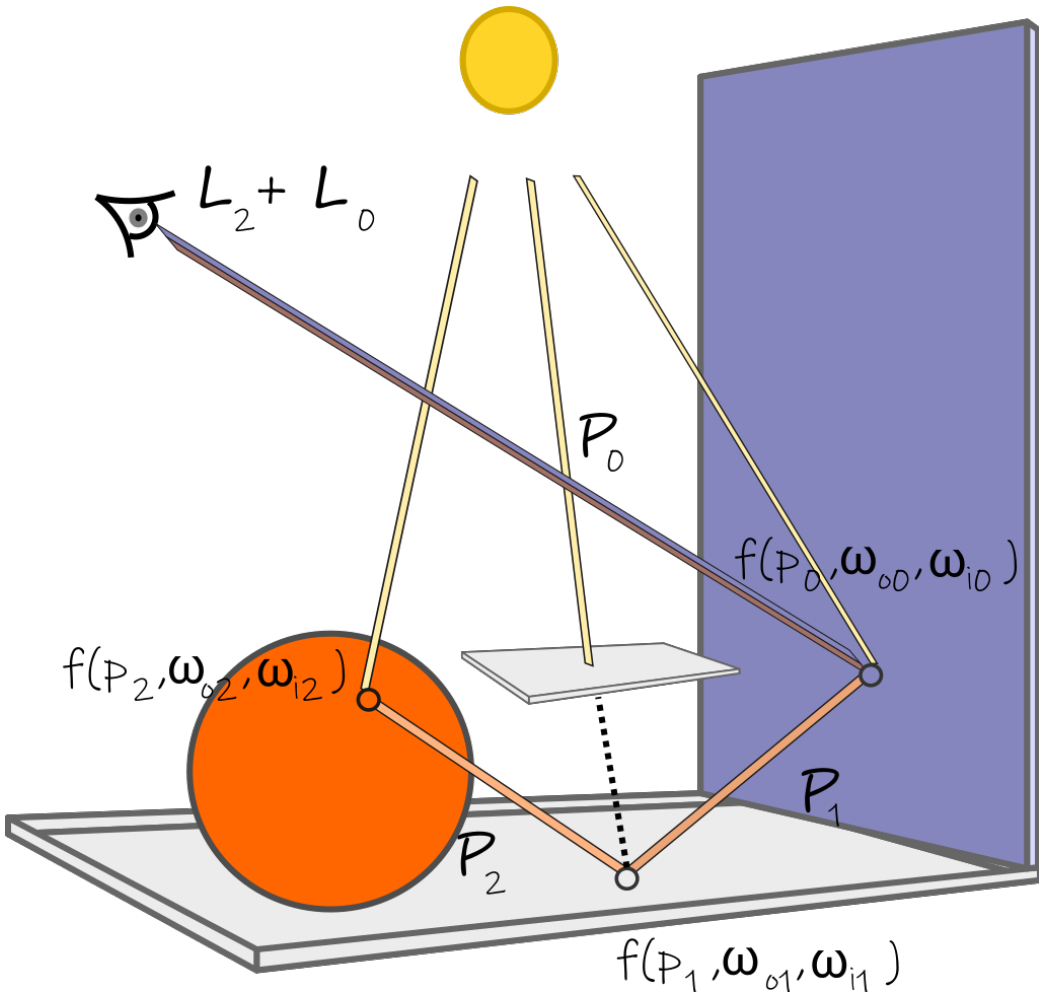



*Figure 3.7: Illustration of evaluating illumination $L$ for a path $\mathcal{P}$ with a local illumination ($\mathcal{P}_0$) and two global illumination bounces ($\mathcal{P}_1$, $\mathcal{P}_2$) using path tracing. $L_0$, local illumination, is easily evaluated by solving the rendering equation for $p_0$, i.e. $L_0 = If(p_0, \omega_{o0}, \omega_{i0}) \mid \cos\theta_0 \mid$. The point $p_1$ is occluded from the lights source and thus $L_1 = 0$. Finally, $L_2$ can is solved by applying reflections in point $p_2$, $p_1$ and $p_2$ to the incoming light $I$. As a result, the total illumination for $\mathcal{P}$ is $L = L_0 + L_2$.*

# Chapter 4

# Photon Tracing

As mentioned in 1.2, photon mapping is a Monte Carlo rendering algorithm presented by Henrik Wann Jensen in *Realistic Image Synthesis using Photon Mapping* [21]. It is a two-pass algorithm, first of which is called *photon tracing*. This is a process of evaluating the path space to generate a data structure referred to as *photon map*. This is done by ray tracing a path of light from the light source through multiple bounces in the scene. Evaluated path represents the progress of a virtual particle known as *photons*. When a photon hits an opaque surface, it is stored into the photon map with related information (e.g., incoming light direction and photon power) after which the tracing is continued based on the surface BRDF as we will discuss in Chapter 4.1. Furthermore, we will examine stored photon attributes in closer detail in Chapter 4.3 and we demonstrate photon mapping process in Figure 4.1.

It is worth noting that due the reverse nature of path space integration in comparison to path tracing, the sampling rate of the path space while using photon tracing is not relative to the resolution of the final image but the amount of traced photons used to sample the contribution of each light source. Thus, the amount of power allocated for each photon is relative to the amount of photons used to sample the light with the total sum of power traced being constant:

$$\Phi = \frac{\Phi_{light}}{N}.$$

This does not negate the natural weakness of noise caused by using Monte Carlo integrator with low sample counts, but resulting noise is noticeable different compared to path tracing. This is explored more later related to filtering in Chapter 7.

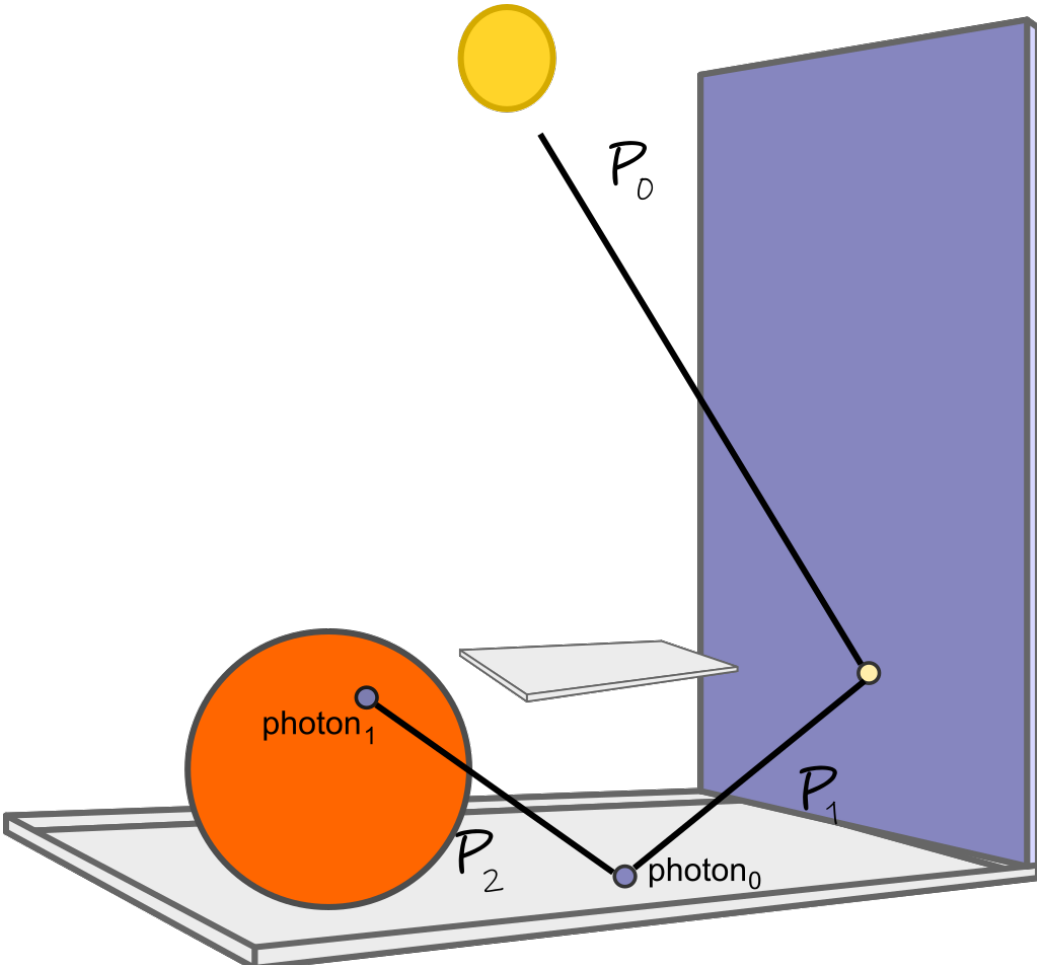



*Figure 4.1: Illustration of evaluating path $\mathcal{P}$ with photon tracing. For the initial bounce $\mathcal{P}_0$ that describes the local illumination the power of the photon $\Phi_0$. However, since photon map is usually generated to represent global illumination, the photon for surface $p_0$ is not stored. For following bounces, $\Phi$ is re-evaluated based on the previous surface reflections, i.e. $\Phi_1 = \Phi_0 f(p_0, \omega_{i0}, \omega_{o0})$ and $\Phi_2 = \Phi_1 f(p_1, \omega_{i1}, \omega_{o1})$.*

If the scene contains multiple light sources, these can be represented by the same photon map while photons are allocated to be traced from different starting locations based on the each light source. In our implementation we define the overall photon count as a constant so that the computational cost is non-dependent on number or intensity of the light sources. Therefore, this global budget is shared with all light sources and the photon count allocations are defined for each lights based on their intensity.

In our work we focused mostly on spot lights as our light source, since this was considered to be the most paramount use case for real-time photon mapping within current performance limitations. However, this is not a restraint of the rendering algorithm as photon mapping can support all common types of light sources including surface emitters. Furthermore, we will discuss several optimization methods that allow photon mapping performance to be acceptable for real-time use. These optimizations include *stratified sampling*, *Russian Roulette* and *reflective shadow maps* (Chapter 5).

## 4.1 Sampling of $\omega_o$ based on surface BRDF

As mentioned above, photon tracing samples the path space represents photons traveling through the scene. However at the start of each path, only the ray describing movement of the photon as it leaves from the light source ($\mathcal{P}_0$ in Figure 4.1) is known. Ray tracing is then used to solve Equation 2.1 for this initial path to find the photon's first surface intersection. But how should we define $\omega_o$ and thus the ray direction of the following bounce? In Figure 3.1, we show $\omega_o$ as mirrored direction of $\omega_i$ on $\omega_n$. Yet, this is only correct if we assume that the surface is a complete mirror, which is a poor representation of most surface materials. Instead, we should define $\omega_o$ based on surface's BRDF.

Defining $\omega_o$ for diffuse reflection is fairly straightforward since diffuse BRDF is sampled by using uniformly distributed directions over the hemisphere (Equation 3.4). However, this can be optimized by using importance sampling called *cosine-weighted sampling* [31, 14.1], where the density is high at the apex of the hemisphere and falls off toward the base. Cosine-weighted sampling can be implemented by applying uniform sampling to a disk located at the normal plane of $\omega_n$ and then projecting those samples to the hemisphere.

In the case of specular reflections, we use cone sampling to sample the specular lobe of the surface, which is defined by BRDF model's normal distribution function (Equation 3.6) and the surface roughness $\alpha$. Cone sampling is explained in Figure 4.2.

We sample the Trowbridge-Reitz GGX distribution to acquire $\omega_m$ by transforming 2D uniformly distributed sample $(u_0, u_1)$ as follows [36]:

$$\theta_h = \arctan\left(\frac{\alpha\sqrt{u_0}}{\sqrt{1-u_0}}\right)$$

$$\varphi_h = 2\pi u_1.$$

Next, by using $\omega_m$ we can define $\omega_o$ as the mirrored incoming direction of $\omega_i$:

$$\omega_o = 2(\omega_m \cdot \omega_i)\omega_m - \omega_i.$$

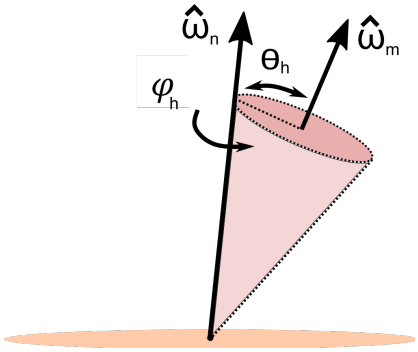



*Figure 4.2: Direction in cone sampling is defined using two spherical coordinates: $\theta_h$ represents the half-apex angle of the cone and $\varphi_h$ the rotation around $\omega_n$.*

## 4.2 Stratified Sampling

Stratified sampling is a sampling method in which we sample from a domain by sampling its partitioned sub-domains. This can be advantageous in real-time application since sampling of different sub-domains can be shared over multiple frames.

We employ stratified sampling to leverage the reasonably low change rate in indirect illumination with a simple scheme: Instead of using the same randomization for every frame, we offset the randomization seed within certain interval. When combining this with offsetting the initial sampling direction from a light source, we can achieve sampling of different sub-spaces within the path space for each iteration. This leads to accumulation of the larger sample set in the path space over multiple frames with each frame sampling different subset of the path space.

Stratified sampling causes time variance in the photon map, but this can be mitigated with temporal accumulation. However, temporally updating the photon map efficiently would be technically challenging and applying the screen space irradiance estimation to the accumulated photon map would significantly increase the cost of the estimation compared to applying it to the subset sampled for each frame. Thus, we apply the screen space estimation only to the subset and accumulate the larger sample set by using temporal filtering (Chapter 7.2) for the estimation result.

## 4.3 Creating a Photon Map from Photon Hits

As a data structure, photon map is extremely straightforward as it is a simple array of photons. These photons contain 32 bytes of data describing the properties of a photon based on a hit surface and attributes of the photon tracing ray. These properties and their packing formats presented in Table 4.1.

| **Property** | **Format** |
|---|---|
| Position | float3 |
| Power | uint - Shared exponent packing |
| Normal | 2xfloat16 - Stereographic packing |
| Light direction | 2xfloat16 - Stereographic packing |
| Ray length | float |
| Sign bits and padding | uint |

*Table 4.1: Data structure for photons. Optimized by aligning structure size to 16 bytes. Most of these properties are packed for performance optimization: The power of the photon is packed using shared exponent packing where three floating point components share the exponent of the floating point presentation while still having unique mantissas. This reduces the size of photon's power component from 16 bytes to 4 bytes with relative small numerical inaccuracy. This is possible since the power is always positive and its components' scales are in relatively same vicinity. Normal and light directions, both being normalized direction vectors, are packed using stereographic projection and thus decreasing the size to 8 bytes.*

Photons are stored in a continuous list. Their location within the list is defined by atomically incrementing a global counter. To optimize the following irradiance estimation, we cull the photons with normals facing away from the camera, as well as the ones that are located outside of the camera frustum. Since we consider photons only as points during the culling, we

incorrectly cull away some of the photons located at near the edges of the view frustum even if the kernels of these photons would still be contributing to the irradiance estimate. This could be possibly prevented by expanding the camera frustum used for the culling. However, this error does not seem to cause any significant visual artifacts when the kernel size in screen space is sufficiently small.

## 4.4 Russian Roulette

Russian Roulette is a common optimization method for Monte-Carlo rendering algorithms. It factors in the surface absorption by terminating the tracing for some photons instead of scaling the power of every photon by the reflectance. This saves a significant amount of ray tracing work while keeping the magnitude of photon power constant over the multiple indirect bounces.

In order to do this, we must solve how to apply the Russian Roulette for surface interaction in relation to photon power. Most of this and following micro surface work was done by Maksim Aizenshtein, my co-author for our article in Ray Tracing Gems [38]. To understand the process let us revisit the rendering equation for Lambertian surfaces:

$$L_o = \int_{S_i^2} \frac{\rho}{\pi} L_i \mid \cos\theta_i \mid \ \mathrm{d}\,\omega_i$$

For an incoming photon hitting a point on surface from the direction $\omega_i$, the radiance is:

$$L_i = E_i \cdot \delta\left(\tilde{\omega}_i - \omega_i\right)$$

Here, $E_i$ is the irradiance of the photon on the virtual surface in front of it, rather than the impact surface. This photon is reflected from the Lambertian surface, and the total irradiance from the point can be written as:

$$E_o = \int_{S_i^2} L_o \cos\theta_o \,\mathrm{d}\,\omega_o \tag{4.1}$$

Normally we would like to sample Equation 4.1 in some way and obtain a Monte-Carlo estimator. However, before doing that, we can introduce a continuous version of a Bernoulli random variable:

$$\begin{gathered} X \sim \mathrm{Ber}\left(\rho\right) \\ f_X\left(x\right) = \left(1-\rho\right) \cdot \delta\left(x\right) + \rho \cdot \delta\left(x-1\right) \end{gathered} \tag{4.2}$$

And use it to rewrite Equation 4.1:

$$\begin{aligned} E_o &= \int_{S_o^2} \int_X \frac{x}{\pi} E_i \cos\theta_i \cos\theta_o \, \mathrm{d}\, F_X(x) \, \mathrm{d}\, \omega_o \\ &= \int_{S_o^2 \cup X} x E_i \cos\theta_i \, \mathrm{d}\, F_{S_o^2, X}(\omega_o, x) \end{aligned} \tag{4.3}$$

The Monte-Carlo estimator for the outgoing power is obtained as summation of the multiple samples within distribution:

$$\Phi_o \approx \Phi_i \frac{1}{N} \sum_j x_j$$

These samples could be considered as photons that are split from the original photon with both power and distribution area divided among these sub-photons. Therefore, given that with enough splitting iterations, the distribution are approaches to zero and the irradiance of the photon can be represented as:

$$E_i = \frac{\Phi_i}{A_i} = \frac{\Phi_i}{A \cos\theta_i}. \tag{4.4}$$

As we can see, the scaling of the photon's power can be replaced with random termination of the photon: If the sample is greater than $\rho$, we cancel the ray and if not, we continue with cosine sampling to trace in a new direction without scaling by the albedo. However, this accounts only for diffuse reflection. How can we expand this for the microfacet specular reflections described in Chapter 3.1.1?

### 4.4.1 Russian Roulette for Reflecting Specular Microfacets

In case of specular microfacet distribution the Russian Roulette process is not as intuitive as it is for surfaces with a flat micro-structure (either Lambertian or specular). First, let's revisit the BRDF Cook-Torrance function for Equation 3.5:

$$f_{\mathrm{CT}} = \frac{D(\omega_m) G(\omega_i, \omega_o, \omega_m) F(\omega_i, \omega_o)}{4 \cos\theta_i \cdot \cos\theta_o}$$

and incoming radiance

$$L_i = E_i \cdot \delta\left(\tilde{\omega}_i - \omega_i\right)$$

Therefore, we can write scattering as

$$\begin{aligned} L_o &= \int_{\Omega_i} L_i f_{\mathrm{CK}} \mid \cos\theta_i \mid \ \mathrm{d}\,\omega_i \\ &= E_i \cdot \frac{D\left(\omega_m\right) G_2\left(\omega_i, \omega_o, \omega_m\right) F\left(\omega_i, \omega_o\right)}{4\cos\theta_o} \end{aligned} \tag{4.5}$$

And the outgoing irradiance:

$$\begin{aligned} E_o &= \int_{S_o^2} L_o \cos\theta_o \,\mathrm{d}\,\omega_o \\ &= E_i \cos\theta_i \int_{S_o^2} \frac{G_1\left(\omega_i, \omega_m\right) D\left(\omega_m\right)}{4\cos\theta_i} \cdot \frac{G_2\left(\omega_i, \omega_o, \omega_m\right) F\left(\omega_i, \omega_o\right)}{G_1\left(\omega_i, \omega_m\right)} \,\mathrm{d}\,\omega_o \end{aligned} \tag{4.6}$$

The density of the scattering direction from visible microfacets (when observed from direction $\omega_i$), as in Heitz et al. [16], is:

$$f_{S_i^2 \mid S_o^2} = \frac{G_1\left(\omega_i, \omega_m\right) D\left(\omega_m\right)}{4\cos\theta_i}$$

Introducing a continuous Bernoulli random variable:

$$\rho\left(\omega_i, \omega_o\right) = \frac{G_2\left(\omega_i, \omega_o, \omega_m\right) F\left(\omega_i, \omega_o\right)}{G_1(S_i^2, S_o^2)}$$

$$X \mid \Omega_i, \Omega_o \sim \mathrm{Ber}\left(\rho\right)$$

$$f_{X \mid \Omega_i, \Omega_o}\left(x, \omega_i, \omega_o\right) = (1-\rho) \cdot \delta\left(x\right) + \rho \cdot \delta\left(x-1\right)$$

And rewriting Equation 4.6:

$$\begin{aligned} E_o &= E_i \cos\theta_i \int_{S_o^2} f_{\Omega_o \mid \Omega_i} \,\mathrm{d}\,\omega_o \int_X f_{X \mid \Omega_o, \Omega_i} \cdot x \,\mathrm{d}\,x \\ &= E_i \cos\theta_i \int_{S_o^2 \cup X} f_{X, \Omega_o \mid \Omega_i} \cdot x \,\mathrm{d}\,\omega_o \,\mathrm{d}\,x \end{aligned} \tag{4.7}$$

With $\omega_o$ and $x$ sampled from their distributions a Monte-Carlo estimator for the outgoing power is obtained:

$$\Phi_o \approx \Phi_i \frac{1}{N} \sum_j x_j \tag{4.8}$$

### 4.4.2 Multichannel Russian Roulette

Since the power of a photon has multiple channels (in the RGB color model), the Bernoulli trial can be modified so that it's done once, instead of per channel. We choose to handle this with the solution described by Jensen [21]:

$$p = \frac{\max\left(\rho_r \Phi_{i,r}, \max\left(\rho_g \Phi_{i,g}, \rho_b \Phi_{i,b}\right)\right)}{\max\left(\Phi_{i,r}, \max\left(\Phi_{i,g}, \Phi_{i,b}\right)\right)}. \tag{4.9}$$

where $p$ is the scalar Bernoulli trial success probability, $\Phi_i$ is the incoming power of the photon, and $\rho$ is the Bernoulli trial success probability per channel. The terms in the sum of the Monte-Carlo estimator for outgoing photon power are then multiplied by $\frac{\rho}{p}$.

We also experiment with per-channel basis Russian Roulette: Instead of terminating the entire photon we would compare the three channels separately and set the failing channels to zero while the photon is terminated only if all three channels are set to zero. This should remove the bias present in the implementation above but it introduced too much noise when the photon density was low.

## 4.5 Transparent surfaces

This far we have addressed only opaque surfaces, which is the traditional case for most of the precomputed global illumination methods. However, one of the main advantages of photon mapping is its fast conversion rate for caustics compared to other Monte Carlo rendering methods, such as ray tracing. Photon mapping, or at least similar algorithms, are actually utilized as a specific rendering pass for caustics in movie production rendering [10]. Therefore, we consider extending the algorithm for transparent surfaces, the most clearly observable case for caustics, to be worth of the effort.

First, we must define if the incoming angle is shallow enough to cause *total internal reflection*. This is the case if the angle is larger than the *critical angle* of the reflection and $n_2 \leq n_1$, $n_1$ and $n_2$ being the *reflective indices* of the participating medias:

$$\theta_c = \arcsin(\frac{n_2}{n_1})$$

For directions $\omega_{ir}$ and $\omega_o$ this can formulated as

$$c_n = \frac{n_2}{n_1}$$

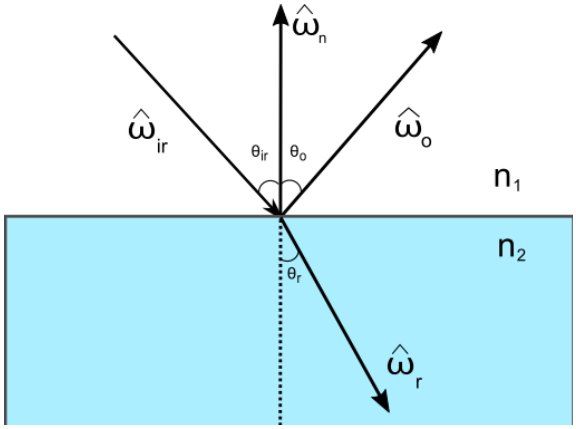



*Figure 4.3: Illustration of incoming $\omega_{ir}$, reflection $\omega_o$ and refraction $\omega_r$ directions in 2D. Please note that the direction of the $\omega_{ir}$ is reversed compared to $\omega_i$.*

$$k = 1 - c_n^2(1 - (\omega_n \cdot \omega_{ir})^2) < 0$$

In HLSL, this check is included in to the *refract* function, which also defines the refraction direction for the ray as

$$\omega_r = c_n\omega_{ir} - (c_n(\omega_n \cdot \omega_{ir}) + \sqrt{k})\omega_n.$$

Furthermore, the reflection direction $\omega_o$ is given by the *reflect* function and is defined as

$$\omega_o = \omega_{ir} - 2(\omega_n \cdot \omega_{ir})\omega_n.$$

If the angle is greater than the critical angle, we assume a mirror-like reflection and continue tracing the photon to $\omega_o$. Since we consider a surface to be a perfect mirror, the power of the photon remains unchanged. If the reflection is within the defined critical angle, we continue by solving Fresnel equations to define the energy ratio between reflection $r_o$ and refraction $r_r$ of the light:

$$r_s = \frac{n_1 \cos\theta_i - n_2 \cos\theta_t}{n_1 \cos\theta_i + n_2 \cos\theta_t}$$

$$r_p = \frac{n_2 \cos\theta_i - n_1 \cos\theta_t}{n_2 \cos\theta_i + n_1 \cos\theta_t}$$

$$r_r = \frac{r_s + r_p}{2}$$

$$r_o = 1 - r_r$$

Since the energy ratio is equal to the ratio of irradiance, we can use this in relation to the photon's power. Therefore, instead of dividing the photon between reflection and refraction, we can use irradiance ratio as probability distribution and then introduce a Bernoulli random variable as earlier with Russian Roulette in Equation 4.2:

$$P_r \sim \mathrm{Ber}(r_r)$$

and with the powers being:

$$E_r = P_r \Phi_i T$$

$$E_o = (1 - P_r)\Phi_i,$$

where T is the transmittance of the transparent surface.

## 4.6 DXR Implementation

Implementing photon tracing using DXR is fairly simple: A ray generation shader is invoked for photons. When using reflective shadow maps (more details in the next chapter), this includes those photons that are terminated by the Russian Roulette during the importance sampling. These empty invocations could be avoided by using indirect dispatch, but unfortunately this is not yet supported for ray generation shaders. The the tracing logic can be done in the ray generation shader while the closest hit shader returns the material parameters of the surface and the ray length. Multiple bounces are implemented by looping the photon tracing in the ray generation shader until we hit the maximum number of bounces or the tracing is terminated by the Russian Roulette.

However, we can further optimize the performance of the tracing significantly: this can be done by moving the Russian Roulette check to hit shader while storing packed the tracing state to the payload. Code snipped for this DXR implementation can be found in Appendix B.

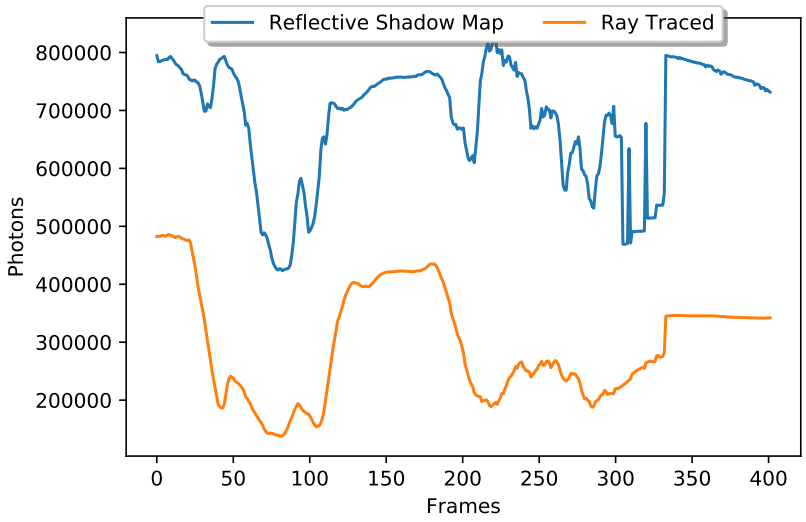
# Chapter 5

# Reflective Shadow Map

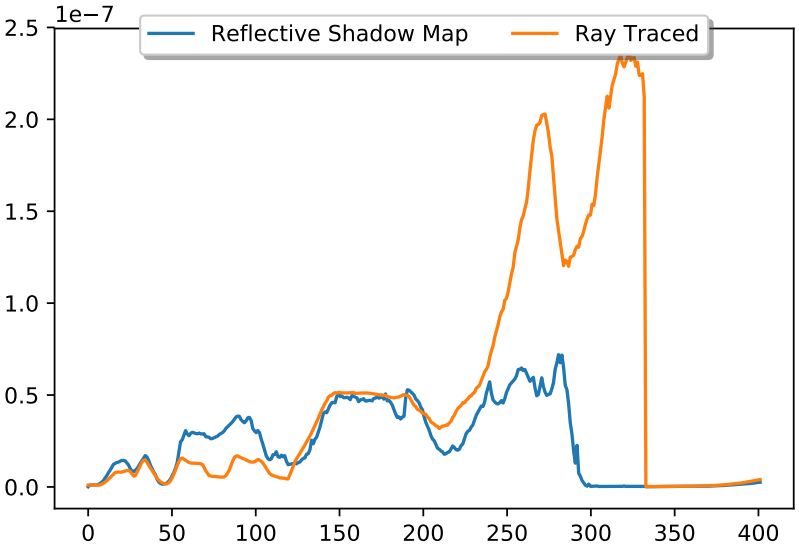
Next we discuss using reflective shadow maps (RSM) to optimize photon mapping for spot lights. RSMs are an optimization method that takes advantage of the fully coherent nature of rays representing photon tracing from the light to the initial surface hit. This coherency makes the initial tracing an optimal use case for the traditional rasterization methods.

RSMs were first introduced by Dachsbacher and Stamminger [13] to gather global illumination of the first indirect bounce by sampling the nearby RSM based on the surface BRDF. However, their work ignore the visibility function, i.e. there is no evaluation if the view from the surface to RSM is blocked by geometry. They later expanded their work to for splatting [14] in which they use the same importance sampling method to define splatting kernel positions as we use to define initial photon sampling positions.

RSMs were introduced in relation to photon mapping by McGuire and Luebke [26]. They utilized the GPU to render the RSM for the direct light and then executing following photon tracing passes on CPU. However, the choice for CPU processing was mostly related to technical limitations at the time — with current access to GPU ray tracing these limitations no longer exist. Furthermore, with the increasing power of the GPU ray tracing, it can be speculated if the relatively small performance gains from RSMs make them worth implementing as it makes the overall photon tracing algorithm much more complex. Yet, we found that RSMs are still valuable since it allows us to use importance sampling (Chapter 3.2) of the path space by defining *sampling points* to replace uniformly distributed photons representing local illumination. This allows a significantly lower number of photons to be terminated by the initial check of the Russian Roulette as well as decreased variance in photons' powers. Analysis for these benefits is shown in Figure 5.1.




*(a) Photon count.*




*(b) Variance.*

*Figure 5.1: Comparison between the number of photons after Russian Roulette and power variance for those photons in 3DMark Port Royal. For this comparison, same number of samples were generated for local illumination using both RSM and uniformly sampling the light cone with tracing. As we can see, RSM retain a significantly larger amount of photons alive while traced samples are killed by Russian Roulette. This is beneficial since we still must pay the computational cost of sampling of the local illumination, which is wasted for terminated samples that contribute nothing the global illumination. Furthermore, there is less variance in power of the surviving photons (averaged of its three channels), which allows smoother illumination result.*

As mentioned before in Chapter 4, the number of photons for light sources is defined as a constant and divided between light sources based on their intensity. Therefore, we are aware how many sampling points is required for their corresponding RSM. Furthermore, it is possible to separate the number of photons from the amount of sampling points by re-using same sampling point, and thus path for local illumination, for several photons while using different randomization seed to generate varying paths for indirect bounces. This becomes useful when photon count is high compared to the resolution of the reflective shadow map.

## 5.1 Drawing Reflective Shadow Maps

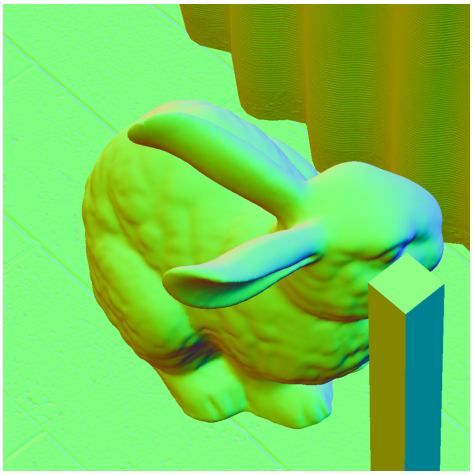
Reflective shadow maps are drawn in the same manner as traditional shadow maps (Chapter 2.3.1). However in addition to the depth, they must contain all necessary surface properties to generate rays for the bounces of indirect light. This is demonstrated in Figure 5.2.

We chose to implement the drawing of RSMs as a separate rendering pass which is executed after rendering of the traditional shadow maps. This was done to make the resolution of the reflective shadow map independent from the traditional shadow map resolution which are commonly desired to be higher. Furthermore, it is common for the resolution of the traditional shadow maps allocated for each light to vary during the runtime depending on e.g. the position of the camera. Therefore, this separation allows us to keep the reflective shadow map size constant and thus we are not required to initialize RSM textures during the runtime or maintain a library of textures with several resolutions. It is possible to use traditional shadow maps as a depth pre-pass where the first depth buffer is used as an optimization to discard surfaces behind it from the beginning of the second draw pass. Unfortunately, without matching resolutions between these passes, the rasterization does not guarantee exactly correct result. However, within our limited test cases this did not cause visible artifacts and might be a valid optimization technique.

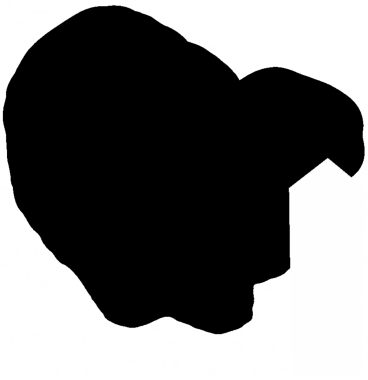
## 5.2 Generating the Probability Map

After completing drawing of the reflective shadow map, we generate a probability map for the first bounce as a function of surface properties and light parameters: First, we compute the probability and Bernoulli trial success for each channel in the same manner as with ray tracing (Equation 4.9). Sec-

*(a) Albedo*



*(b) Normal*



*(c) Metalness*

*(d) Roughness*

*Figure 5.2: RSM textures: albedo, normal, roughness and metalness maps. These parameters are packed into to two textures: albedo (rgb) and roughness (a) is packed to r8g8b8a8-unorm format while another texture with r16r16b16a16 format contains normal (rgb) and metalness (a). Furthermore, the normal texture contains a bit representing if the surface is opaque or transparent.*

ond, we multiply the absorption probability by light falloff. Third, we allow additional artist parameters that allow some artistic control over the distribution of the computational resources and therefore the quality of illumination estimation.

These artist parameters define separate near and far planes for the photon mapping within light's near and far planes and we set the probability for surfaces outside this range to zero. This allows us to not to waste photons on

the surfaces that will have minimal impact on the visible global illumination. This has been illustrated in Figure 5.3.

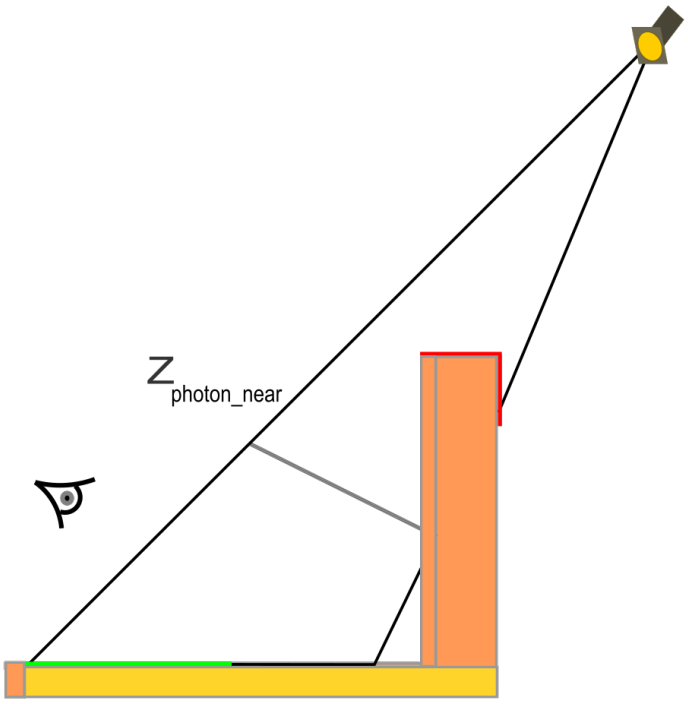



*Figure 5.3: A simple example of artist controls for RSM clipping planes. It can be safely assumed that the contribution of photons hitting surfaces on outer portion of the wall, marked with red, will be minimal for the visible illumination. Therefore, we concentrate photons to more valuable surfaces, marked with green, by setting the probability to zero for surfaces closer than the clipping plane* $Z_{photon\_near}$.

## 5.3 Importance Sampling of Ray Casting Positions for Indirect Light

It is worth noting that probability map we generated in Chapter 5.2 is a pixelwise probability that is not normalized or accumulated in screen space. Naive sampling of RSM's probability function would require both of these operations. However, they are non-trivial operations to in terms of GPU parallelization and thus we choose to avoid them. Instead we apply a technique called wavelet importance sampling introduced by Clarberg et al. [11].

Finally, we store the sampling points to be used in following photon tracing pass.

### 5.3.1 Wavelet importance sampling

Wavelet importance sampling is a two step algorithm: First, we apply discrete Haar wavelet transform to the probability map. Second, we reconstruct the signal for each sample location in low discrepancy sequence and wrap the sampling positions based on the scaling coefficient of each iteration in a wavelet transformation.

#### 5.3.1.1 Haar Wavelets

Haar wavelets are a set of wavelet functions that can be used as a hierarchical function analysis method. Using this set in unison with a scaling function allows us to represent a target function over an interval in terms of orthonormal basis. Haar wavelet mother functions $\psi(x)$ can be described as

$$\psi(x) \equiv \begin{cases} 1 & 0 \leq x < \frac{1}{2} \\ -1 & \frac{1}{2} < x \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

and scaling function $\phi(x)$

$$\phi(x) = \begin{cases} 1 & 0 \leq t < 1 \\ 0 & \text{otherwise} \end{cases}$$

Furthermore, the normalized basis functions are:

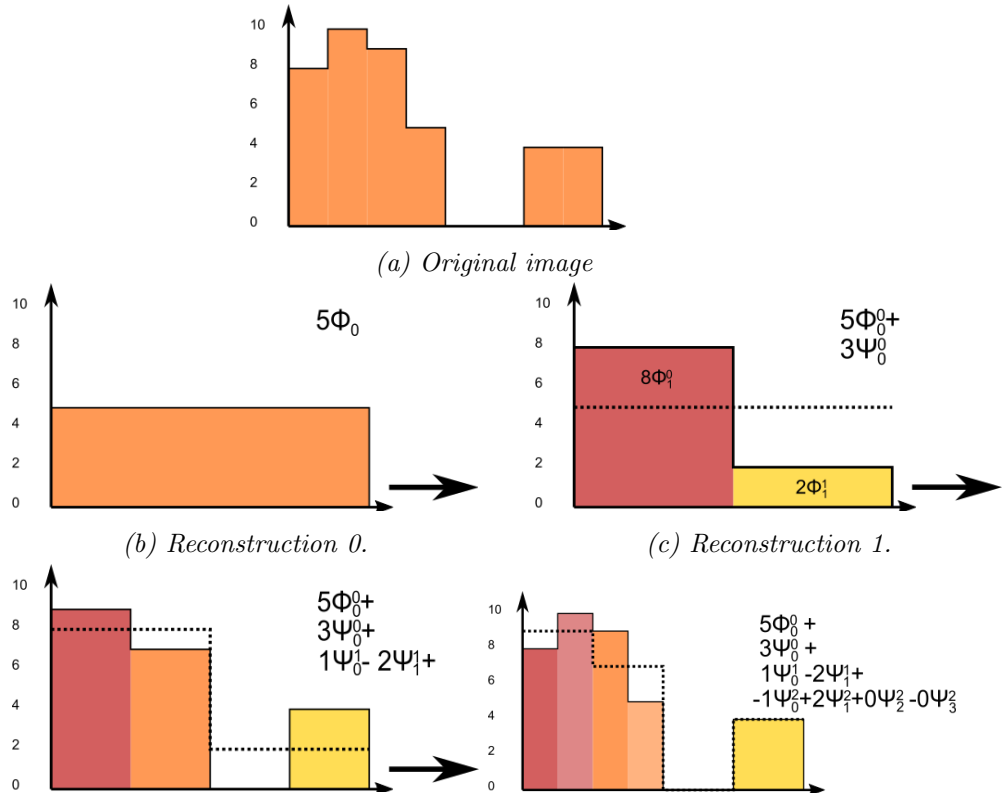
$$\phi_t^l(x) := 2^{I/2}\phi\left(2^l x - t\right),$$
$$\psi_t^l(x) := 2^{I/2}\psi\left(2^l x - t\right),$$

where $l$ is the level within the hierarchy and $t$ the translation of the functions. Given this, image can be described as

$$H(x) = H_{0,0}^0\phi_0^0 + \sum_I \sum_t H_{t,1}^l \psi_t^l = \sum_i H_i \Psi_i.$$

As an example, we show a reconstruction of a 1D image in Figure 5.4.
Probability for each level of the hierarchy is therefore defined as:

$$P_i^l = \frac{H_{1,0}^i}{\sum_{\mathrm{t}} H_{\mathrm{t},0}^l} \tag{5.1}$$




*(a) Original image*

*(b) Reconstruction 0.*

*(c) Reconstruction 1.*

*(d) Reconstruction 2.*

*(e) Reconstruction 3 - original image.*

*Figure 5.4: Reconstruction process of the wavelet for 1D image: By adding the every levels' wavelet function to the base level scaling function, we can reconstruct the original image. In 5.4c we can also see that higher level scaling functions are simple combination of lower level's scaling and wavelet function, i.e.* $8\phi_1^0 = 5\phi_0^0 + 3\psi_1^0$. *In this example we are using original image* $[8, 10, 9, 5, 0, 0, 4, 4]$ *and its Haar representation is* $[5, 3, -2, -1, 2, 0, 0]$. *Reconstruction follows as* $[5] \rightarrow [5+3, 5-3] = [8, 2]$ *and so on. Figure and the example are based on the original importance sampling work by Clarberg et al. [11].*

#### 5.3.1.2 Discrete Wavelet Transform of 2D Image

To compute a discrete wavelet transform we implemented 2D Haar wavelet transform for a discrete 2D image as a multi-pass filtering process that consist of $\log_2(n)$ transformation steps with $n$ being the dimensional resolution of the original image. These steps consist of a high-pass filter applied to a two-by-two pixel quad that defines the three *detail coefficients* and a *scaling coefficient*. The scaling coefficient is a low-pass filtered result of square and the detail coefficients define the difference between original pixels and the scaling coefficient. We show result of the this in Figure 5.5

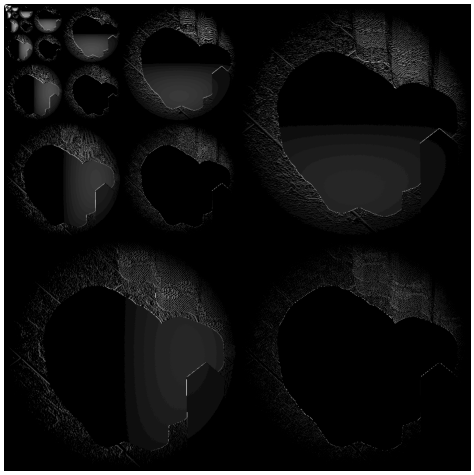


*Figure 5.5: Discrete wavelet transform of the probability function.*

Unfortunately, the wavelet transformation must be iterated for the entire hierarchy so that the final resolution of the scaling coefficients is only two-by-two. This is because the scale of offsetting in following wrapping is inversely proportional to the increase of the contributing hierarchy level. However, launching individual compute shader passes for low kernel dimension is sub-optimal and thus we implemented a separate computer shader pass for lower resolution transformation that uses shared memory similarly to any standard reduction implementation.

### 5.3.2 Hierarchical Wrapping of the Sampling Points

Hierarchical wrapping is a method to offset a set of sampling points based on the wavelet representation of probability function. First, we must generate the original sampling positions. These are samples generated from a low-discrepancy sequence, namely a Hammersley set. Second, we must sample the hierarchical probability using Equation 5.1 for that sample position. Third, we offset the sample within the scale of the level in the hierarchy. This process is demonstrated in Figure 5.6. This offsetting is applied for every hierarchy level of the wavelet transformation.

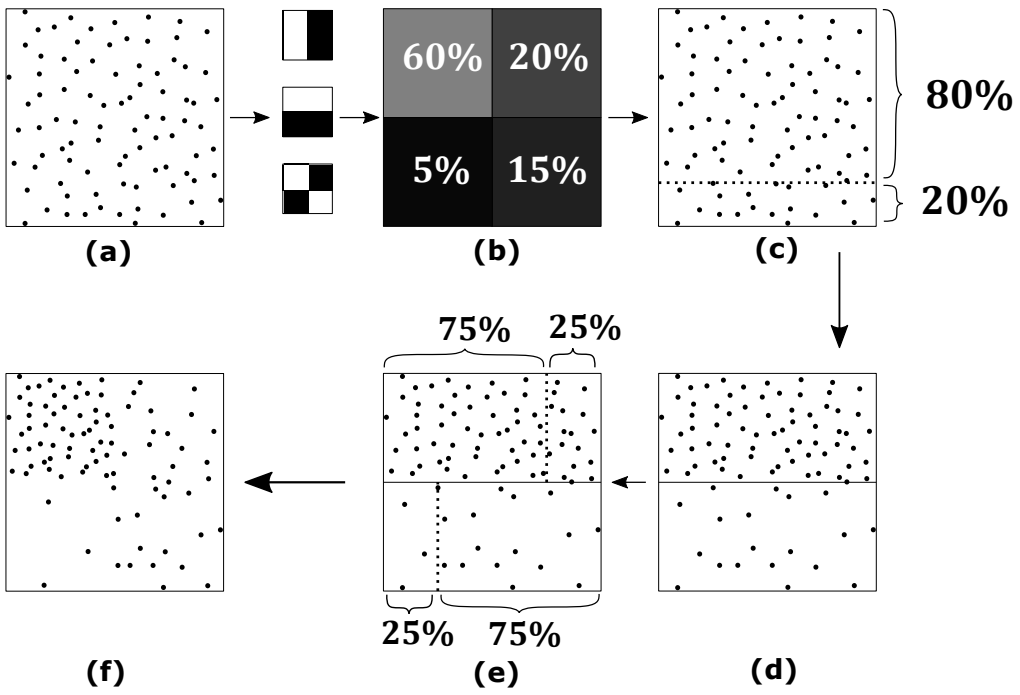



*Figure 5.6: Wrapping a set of sampling positions by an iteration of the wavelet transformation. The initial sampling positions (a) are wrapped first in horizontally (c) and then vertically (e) using the rations of the scaling coefficients in the active quad (b). (Illustration after Clarberg et al. [11].)*

### 5.3.3 Storing the Sampling Points

After generating sampling positions, we run Russian Roulette optimization before storing them. It is worth noting that each sample is completely non-dependent on other samples and thus there is no need for synchronization except an atomic counter used to define a storing location in the output buffer. However, because this location in the buffer changes between frames, we must generate necessary randomization seeds now using the sampling index instead of later in photon tracing using invocation index that is also invocation's sample buffer location. This is done to keep photon paths deterministic between frames. Sampling positions are stored using a data structure presented in Table 5.1.

| Property | Format |
|---|---|
| Position | float3 |
| Power | uint - Shared exponent packing |
| Normal | 2xfloat16 - X and Y components |
| Normal | float16 - Z component + padding |
| Seed | uint |
| Padding | uint |

*Table 5.1: Data structure for sampling points. Aligned to 16 bytes to make it more suitable for some hardware.*

Since we are doing importance sampling to define to sampling locations, we must scale the the photon power to make the sampling unbiased. I.e, since we sample on locations that have higher reflectance, and thus more probable to survive Russian Roulette, we must balance photons' power to make the total amount of light to remain the same. Fortunately, we know the mean of RSM probabilities based on the lowest level of wavelet transformation function. Therefore, we scale the photon power as

$$\Phi_{scaled} = \frac{P_0}{P}\,\Phi_{uniform}$$

with $P$ being defined in Equation 5.1. However, this assumes that scaling should be based on the entire RSM and most of the light shapes are cones instead frustums and therefore do not inhabit the entire area of the square-shaped texture in RSM. This can be solved by simply adjusting the scaling based on the ratio between areas of a rectangle and a circle:

$$\Phi_{scaled} = \frac{P_0}{P}\,\frac{\pi r^2}{(2r)^2}\,\Phi_{uniform} = \frac{P_0}{P}\,\frac{\pi}{4}\,\Phi_{uniform}$$

## 5.4 Markov Chain Monte Carlo with Hamiltonian Probability Function for RSMs

As explained in the previous chapter, the wavelet importance sampling applies importance sampling for paths from the light to the first surface reflectance. However, would it be possible to utilize importance sampling for the following indirect bounces? Unfortunately, the completely incoherent nature of the indirect bounces makes this challenging. We choose to approach

this problem using a *Markov Chain Monte Carlo* (MCMC) sampling of the RMS.

Our solution is inspired by *Metropolis Light Transport* (MLT) introduced by Veach et al. [42] which uses *Metropolis-Hastings sampling* to explore the path space. However, unlike MLT our solution does not discard any samples and thus it is more alike to *Gibbs sampling*. Also, MLT generates proposals for entire path by mutating previously evaluated paths, whereas we only mutate the sampling points within RSM. Furthermore, there has been some previous work related to using MCMC for sampling of RSM, such as work of Barak et a. [8] that used MCMC to determine locations of virtual point lights.

These concepts are alluded in this chapter, but given this being quite minor part of our work and the extensive scope of stochastic computation methods, the discussion is kept quite limited.

### 5.4.1 Generating Markov Chain Monte Carlo for RSM

MCMC methods simulates a distribution function by producing an ergodic *Markov Chain* to utilize statistical inference to model the distribution based on the experimental data. Target distribution for this modeling is referred to as *equilibrium distribution*. This Markov Chain is described by a *Markov Matrix* $M$, which we generate for RSM's probability function during the photon tracing and use it for the following frame.

Let us examine generating a Markov matrix for RSM using 1D image as an example. This works exactly in the same manner for 2D image, but reducing dimensions make the example easier to follow:

A 1D image with $n$ pixels and the pixelwise probability function $p$ results $M$ being a $n \times n$ dimensional matrix. Fortunately, $M$ is a diagonal matrix as each pixel affects only its own probability:

$$Mp = \begin{bmatrix} m_{11} & 0 & \dots & 0 \\ 0 & m_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & m_{nn} \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix} = \begin{bmatrix} m_1 p_1 \\ m_2 p_2 \\ \vdots \\ m_n p_n \end{bmatrix}$$

Therefore, this allows us to store $M$ in a texture with the same resolution as the original image.

During the photon tracing we atomically increment $M$ so that the incremented pixel location is defined by the sampling position. This is done for each photon bounces using a constant value with this value being positive for photon within the camera frustum and negative for photons outside of it. Unfortunately, due to atomic operations on GPU being limited to integers, these constants and the texture storing $M$ are in integer format.

### 5.4.2 Applying Markov Matrix to probability function of RSM

In previous section we explain the generation a Markov matrix, but applying it to the probability function can be slightly challenging: First, due restrictions mentioned previously, values of $M$ are in integer format, whereas the probability function has normalized floating point values. Second, we must account to the sparse nature of $M$ as we would prefer the photons contributing to a pixel neighborhood instead of a single pixel. This could be solved by using a filtering function, such as a Gaussian blur, but these filters come with noticeable performance cost. We solved these issues by applying $M$ as a force to a probability function model after 1D Hamiltonian mechanics. Therefore, this probability function contains *velocity* and *position* parameters, that are changed by acceleration defined using Markov matrix, Hooke's law and drag as follows:

$$a = mC_m - kx_{i-1} - v_{i-1}C_d$$

with $C_m$ being a constant transforming integer value of $m$ to a floating point force value, $k$ as a spring constant and $C_d$ a drag constant. Velocities and positions are updated as:

$$v_i = v_{i-1} + a$$

$$x_i = \text{clamp}(0, 1, x_{i-1} + v_{i-1})$$

This is then applied to the probability function using a simple multiplication. However, we prevent the probability being zero and therefore the final probability $p_{final}$ being:

$$p_{final} = \frac{1}{10} + \frac{9}{10}x_i p$$

By itself this does not solve the issues with sparsity of $M$, but combining it with stratified sampling (Chapter 4.2) allows fairly good results.

# Chapter 6

# Screen Space Irradiance Estimation

With the increase of ray tracing potential, the efficiency of the photon shading becomes essential for the overall performance. This is the process of generating screen space irradiance estimation based on the *distribution kernel* of each photon in the photon map. There are two principal approaches — *scattering* and *gathering*. Mara et al. [25] provided a conclusive overview of both approaches and we experimented with two of these solutions:

**Scattering** We implemented a scattering method of using graphics pipeline to draw the kernel while accumulating the results with GPU's blending unit.

**Gathering** We experimented with a gathering method which uses compute shaders to apply tile-based culling for photons and then do pixelwise gathering to accumulate the contribution utilizing the culling result as an acceleration structure.

In our testing the splatting proved to be more efficient, but gathering has also some potential and we provide a more detailed comparison in Chapter 6.4.

## 6.1 Defining the Splatting Kernel

Selecting a good kernel size for each photon is important: if the kernels are too wide, the lighting will be excessively blurry, and if they are too narrow, it will be blotchy. It is particularly important to avoid too-wide kernels because a wider kernel makes a photon cover more pixels and thus leads to

more rasterization, shading, and blending work per photon. Furthermore, incorrect kernel selection can cause several types of biases and errors [35]; minimization of these has been the focus of a substantial amount of research.

In our approach, we start with a spherical kernel and then apply a number of modifications to it in order to minimize various types of error. These modifications can be categorized into two main types: uniform scaling and modification of the kernel's shape.

### 6.1.1 Uniform Scaling of the Kernel

Uniform scaling of the kernel is a product of two terms, the first one based on the ray length and the second on an estimation of the photon density distribution.

**Ray Length** We scale the kernel according to the ray length using linear interpolation to a constant maximum length. This method is an approximation of the ray differential and can be interpreted as treating the photon traveling along a ray cone instead of a ray and factoring in the growth of the cone base as its height increases. Also, we can assume lower photon densities as the ray length increases, since it is probable that photons scatter to a larger world-space volume. Thus, we want a relatively wide kernel in that case. The scaling factor is

$$s_l = \min\left(\frac{l}{l_{\max}},\ 1\right), \tag{6.1}$$

where $l$ is the ray length and $l_{\max}$ is a constant defining the maximum ray length. However, $l_{\max}$ is not required to be the maximum length of the rays cast during photon tracing but instead the length that we consider to be the maximum height of the cone. This constant should be related to the overall scale of the scene and can be derived from its bounding box.

**Photon Density** We would like to further scale each photon's kernel based on the local photon density around it: the more photons that are nearby, the smaller the kernel can (and should) be. The challenge is efficiently determining how many photons are near each one since they are stored only as a point cloud. Therefore, we apply the simple approximation by maintaining a counter during the photon tracing pass for each screen-space *tile*. Tiles are a sparser representation of the screen space containing multiple neighboring pixels. When a photon is deposited in a tile, the counter is atomically incremented. This is obviously a crude approximation of the density function, but

it seems to produce fairly good results. Alternatively, we experimented using temporal density data from previous frame's filtered density function, but this came with a noticeable performance cost and there were no significant quality gains. Thus, we opted to use tile-based solution.

We then apply the density-based scaling as a function of the tile's area in view space:

$$a_{\text{view}} = z_{\text{view}}^2 \frac{\tan(\alpha_x/2)\tan(\alpha_y/2)\ t_x\, t_y}{r_x\, r_y},$$

where $\alpha_x$ and $\alpha_y$ are the apertures of the camera frustum, $z_{\text{view}}$ is the distance from the camera, $t_x$ and $t_y$ are the tile dimensions in pixels, and $r_x$ and $r_y$ represent the image's resolution. In most cases a tile does not have a uniform depth, so we use the depth of the photon position. Most of this arithmetic can be precalculated and replaced with a camera constant:

$$c_{\text{tile}} = \frac{\tan(\alpha_x/2)\tan(\alpha_y/2)\ t_x\, t_y}{r_x\, r_y},$$

$$a_{\text{view}} = z_{\text{view}}^2\, c_{\text{tile}}.$$

Thus, scaling the circular kernel to have the same area in the view space as the tile can be calculated as

$$a_{\text{view}} = \pi r^2 n_p, \quad r = \sqrt{\frac{z_{\text{view}}^2\, c_{\text{tile}}}{\pi n_p}},$$

where $n_p$ is the number of photons in the tile. This value is clamped to remove any extreme cases and then multiplied by the constant $n_{\text{tile}}$, which is equal to the number of photons that we expect to contribute to each pixel:

$$s_d = \text{clamp}(r, r_{\min}, r_{\max})\, n_{\text{tile}}. \tag{6.2}$$

### 6.1.2 Adjusting the Kernel's Shape

We can further improve the reconstructed result by adjusting the kernel's shape. We consider two factors. First, we decrease the radius of the kernel in the direction of the surface normal of the intersected surface. Second, we scale the kernel in the direction of the light in order to model the projected area that it covers on the surface. This results in the kernel being a tri-axial ellipsoid, which has one axis, $\mathbf{n}$, that has the direction $\omega_g$ of the normal. The other two axes are placed on a tangent plane defined by the photon normal, referred to as the *kernel plane*. The first of the two, $\mathbf{u}$, has the direction of

$\omega_i$ projected onto the kernel plane, while the second, $\mathbf{t}$, is orthogonal to it and in the same plane. This vector basis is illustrated in Figure 6.1.

The magnitude of $\mathbf{n}$ is $s_n s_l s_d$, where $s_n$ is a constant that compresses the kernel along the normal so that it is closer to the surface. This is a common approach: it was done by Jensen [21] for gathering with a varying gathering radius and by McGuire and Luebke [26] with compression of their splatting kernel along the normal. Compared to a spherical kernel, this provides a better approximation of the surface. However, if the kernel is compressed too much, the distribution on objects with complex shapes or significant surface curvatures becomes inaccurate, as the kernel disregards samples farther away from its plane. This can be compensated for by making the magnitude be a function of the surface curvature, but in our implementation this factor is constant.

The magnitude of $\mathbf{u}$ is $s_u s_l s_d$, where $s_u$ is defined as a function of the cosine of the angle between the hit normal and the light direction:

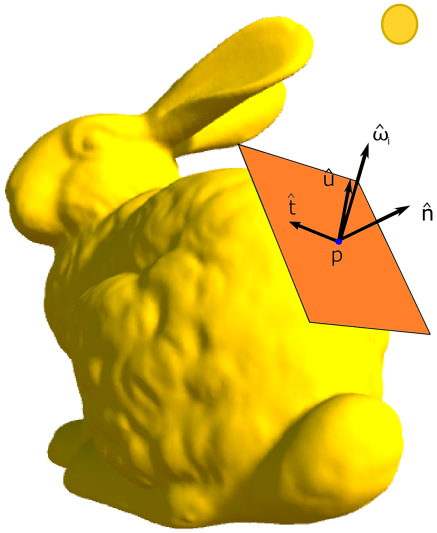
$$s_u = \min\left(\frac{1}{\omega_g \cdot \omega_i},\ s_{\max}\right), \tag{6.3}$$

where $s_{\max}$ is a constant defining the maximum scaling factor. Otherwise, the magnitude would approach infinity as the angle between $\omega_g$ and $\omega_i$ decreases to zero. Intuition for this approach originates in ray differentials and the ray cone representation of the photon: as the incoming direction of the photon becomes orthogonal to the normal direction of the surface, the area of the base of the cone that is projected onto the kernel plane increases. Finally, the magnitude of $\mathbf{t}$ is $s_l\ s_d$.

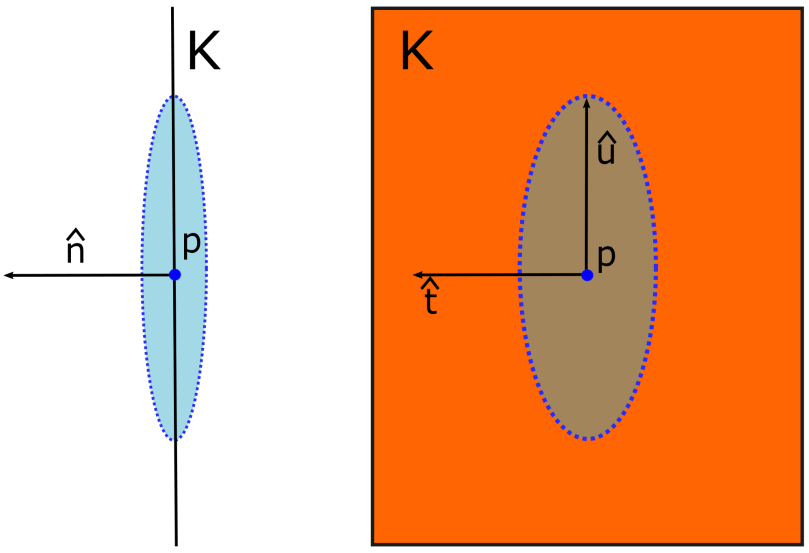
## 6.2 Photon Splatting

Photon splatting is a point-based rendering method that distributes radiance of the photon across pixels in screen space. This is done by using a graphics pipeline to draw a splatting kernel for each photon while using additive blending to sum up results of multiple overlapping kernels. This is implemented as an instanced indirect draw of a sphere approximation (*icosahedron*) while using the atomic counter that we used previously to define a photon storing location as an indirect argument to designate number of photons to draw.

To apply the kernel shape discussed in Chapter 6.1, we apply a transformation to the vertices in a vertex shader and since the original kernel is a sphere, we can assume the coordinate frame of the kernel's object space to be the coordinate frame of the world space. We separate the vertex position to the base vector of the kernel space and they are scaled as follows:




*(a) Base vectors of the kernel plane.*




*(b) Shaping kernel on the kernel plane.*

*Figure 6.1: 6.1a: $\omega_g$ is aligned to the photon normal $\widehat{\mathbf{n}}$, which also defines the kernel plane $k$. Two other basis vectors lie in $\mathcal{K}$ such that $\widehat{\mathbf{u}}$ is the projection of light direction $\omega_i$ on to the kernel plane and $\widehat{\mathbf{t}}$ is orthogonal to $\widehat{\mathbf{u}}$. in 6.1b the shape of the kernel is modified by scaling along those vectors.*

$$\mathbf{v}_{kernel} = \begin{bmatrix} \mathbf{n} & \mathbf{u} & \mathbf{t} \end{bmatrix} \begin{bmatrix} \widehat{\mathbf{n}}^\intercal \\ \widehat{\mathbf{u}}^\intercal \\ \widehat{\mathbf{t}}^\intercal \end{bmatrix} \mathbf{v} \tag{6.4}$$

We choose to keep the pixel shader for splatting kernel as simple as possible as it can easily become a performance bottleneck. Thus, only operation here is a depth check to define if the surface that we are calculating radiance for is within the kernel. We used two approaches to solve this: One solution is to apply a clipping operation with a simple depth check between difference of linearized depths of the surface and kernel plane against a constant value that is scaled by the kernel compression constant. While highly efficient in terms of performance, this approach is rather crude and can lead to some significant artifacts. As an alternative, we can test if the pixel is within kernel by using the *standard equation* of tri-axial ellipsoid. This is defined formally as:

$$\frac{x_{\mathcal{K}}^2}{\|\mathbf{u}\|^2} + \frac{y_{\mathcal{K}}^2}{\|\mathbf{n}\|^2} + \frac{z_{\mathcal{K}}^2}{\|\mathbf{t}\|^2} \leq 1, \tag{6.5}$$

while $x_{\mathcal{K}}$, $y_{\mathcal{K}}$ and $z_{\mathcal{K}}$ are the pixel's position in kernel space. We provide more details comparing the two methods above in Chapter 6.4. Finally, we apply the kernel with uniform distribution to the splatting result with the irradiance of pixel being

$$E_i = \frac{\Phi}{A}, \tag{6.6}$$

where $A$ is the area of the ellipse

$$A = \pi\|\mathbf{u}\| \cdot \|\mathbf{t}\| = \pi(S_l \cdot S_d)(S_l \cdot S_d \cdot S_u)$$

It is worth noting that irradiance here is scaled by the cosine term and thus implicitly includes information from the geometry normals.

For accumulation of irradiance we use half-precision floating-point format (per channel) in order to avoid numerical issues with lower bit formats. Furthermore, we also accumulate the average light direction as a weighted sum with half precision floats. The motivation for the latter is discussed in Chapter 7.4.

As mentioned before, we are currently using traditional additive blending (summation of blended surface values) to accumulate the contributions of multiple kernels. However, this might not be the most optimal solution as in blending the pixel order is sorted by the render output unit. This can cause significant overhead. As a potential optimization there are vendor

specific extensions which allow unordered atomic writes with half precision formats. Also, it could be possible to handle the synchronization by hand but this would require creating two critical sections to the shader and thus was not something we considered worth testing. Nevertheless, this is a major bottleneck for the splatting that will require future work.

Sample code for the photon splatting is presented in Appendix C.

### 6.2.1 Optimization of Splatting by Using Lower Resolution

Splatting can be an expensive process, which is especially the case when rendering in higher resolutions. Thus, we choose to optimize this by lowering the resolution of the splatting to half of the native rendering resolution. This did not cause a noticeable decrease in visual quality for the final result. It is possible to use even quarter resolution, but this causes some precision issues.

However, using lower resolution requires a change to the depth clipping in the pixel shader to remove any irradiance bleeding between surfaces: The half-resolution depth stencil used for stencil drawing should be downscaled using the closest pixel to the camera. However, the depth used in pixel shader clipping should be downscaled as the farthest pixel from the camera. As a result, we draw the splatting kernel for only pixels that are entirely within the kernel in full resolution. This causes jagged edges in the splatting result, but they will be removed by the filtering.

## 6.3 Tile-Based Photon Gathering

The tile-based gathering is a compute shader based method to solve the irradiance estimation. It was introduced by Mara et al. [25] and it is based on using tiles as an acceleration structure. First, the algorithm includes a tile-culling step where we define a set of photons contributing to each tile. This is very similar to common light-culling techniques used in deferred rendering. However, in contrast to light culling where the maximum amount of lights contributing to a tile can be reasonable estimated in order to allocate a buffer with constant size, the variance in the number of photons per tile makes this approach impractical for gathering. To solve this we must use dynamical allocation within the buffer. Unfortunately, implementing this requires multiple compute passes. Finally, we apply a gathering pass to compute pixel wise contribution of each photon within the tile. All in all, this process requires five compute shader passes:

1. Indirect Argument
2. Photon Count
3. Allocation
4. Copy
5. Gathering

**Indirect Argument** As the number of photons is unknown during the recording of the command list on CPU, we must execute a simple compute pass the create indirect argument for the two following passes that are executed for each photon.

**Photon Count** We execute an indirect compute pass where we apply tile-based culling to the kernel against the neighborhood of tiles where the photon is located in. We atomically increment a counter for the culled tiles and thus acquire exact number of photon contributing to each tile.

**Allocation** Based on the counting results, we define every tile's offset from the begin of buffer by using an atomic counter.

**Copy** As before, we launch indirect compute pass with invocations for all photons and apply the tile-based culling. Then we assign culled photon's index to the tiles and store those indices to another buffer. This means that we do not have to copy any of the photon data, which is significantly much larger than an integer used for indexing. Therefor, we can avoid both the bandwidth cost caused by the rearranging of photon data and the noticeable memory overhead caused by photons contributing to several tiles. However, the downside is that this causes accessing to photon data to be incoherent since it must be done through the indirect indexing buffer. It is worth noting that since we do not know the amount of tiles each photon contributes, we can not explicitly define size for the indirect indexing buffer and thus are required to estimate. If the estimation is too low, this causes clearly visible corruption to the irradiance result.

**Gathering** We start by loading tile's photon data to shared memory as it is is shared by all of the pixel within the tile. Then we loop over these photons, transform the pixel position to the coordinate system of the distribution kernel and define if the pixel is within the ellipsoid using its standard equation

(Equation 6.5). If the pixel is within the kernel, we add its contribution to the pixel's irradiance and weighted light direction.

## 6.4 Comparison of Scattering and Gathering methods

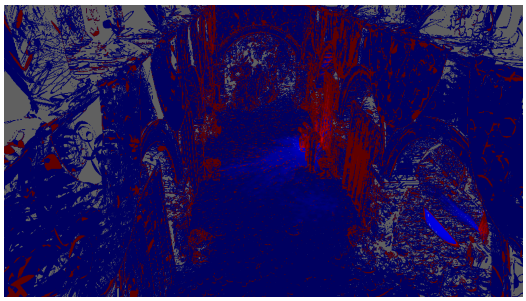
As we saw in Chapter 6.3, the implementation for gathering becomes significantly more complex compared splatting and requires careful handling of several different atomic counters. Furthermore, in our test cases its performance was significantly worse compared to splatting (Table 6.1). However, our tile-culling was done using a sphere defined around the kernel and depending on the shape of the kernel, this can cause a significant amount of false positives. Thus, more advanced culling could achieve better results. Furthermore, gathering is advantageous in cases where we would like to read more surface attributes than just depth, such as the one we will discuss in Chapter 7.4. In Figure 6.2 we show the difference comparisons between different clipping methods for splatting as well as differences between splatting and gathering.

| **Scene** | **Photons** | Splatting (el.) | Gathering | Splatting (lin.) |
|---|---|---|---|---|
| Sponza | 500000 | 1.3 | 6.6 | 1.5 |
| Conference room | 100000 | 0.9 | 7.4 | 1.1 |

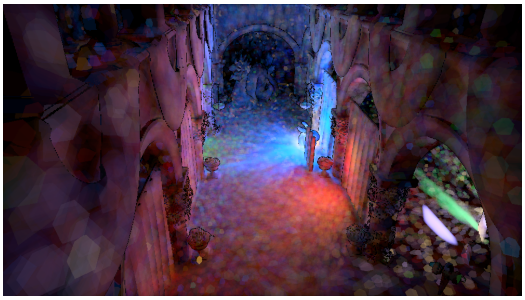
*Table 6.1: Performance comparison between different screen space irradiance estimation methods. As we can see, despite additional computation comparing with ellipsoid can be more efficient since it has more aggressive culling, and thus less blending, compared to linear depth clipping.*

*(a) Splatting with ellipsoid check.*

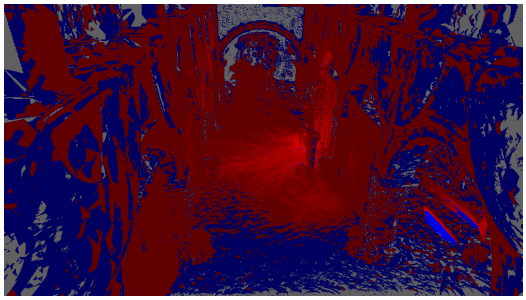
*(b) Gathering.*



*(c) Difference splatting and gathering.*



*(d) Splatting linear depth check.*



*(e) Difference with linear and ellipsoid checks.*

*Figure 6.2: Comparisons between different screen space irradiance estimation methods.*

# Chapter 7

# Filtering

As typical for real-time Monte-Carlo rendering methods, compensation of the low sample count by filtering is essential to achieve high-quality results and there has been significant advances in the field of denoising within the recent years. However, the noise caused by photon distribution kernels is quite different from the high-frequency noise caused by path tracing that has been the main focus of these denoising efforts. Thus, a different solution was required.

We choose to approach this by using both temporal and spatial accumulation of samples with geometry-based edge-stopping functions. This is similar to previous work by Dammertz et al. [15] and Schied et al. [33] with our implementation using edge-avoiding À-Trous wavelet transform for spatial filtering. Filtering can be implemented in lower resolution to decrease the computation cost, but we opted not to since it caused artifacts due to G-buffer discrepancies.

## 7.1 Edge-Stopping Functions

These functions, from the work of Schied et al. [33], are aimed to prevent filtering over geometrical boundaries by generating normalized weights based on the surface attributes of current pixel $p$ and the sample pixel $q$. First, we compare the depth differences:

$$w_z\left(p,q\right)=\exp\left(-\frac{|z(p)-z(q)|}{\sigma_z\left|\nabla z\left(p\right)\cdot\left(p-q\right)\right|+\varepsilon}\right), \tag{7.1}$$

where $\nabla z\left(p\right)$ is the depth gradient and $\sigma_z=1$ is constant value defined by experimentation. Second, we take into account the difference of the surface normals:

$$w_n(p,q)=\max(0,\ n(p)\cdot n(q))^{\sigma_n}, \tag{7.2}$$

where $\sigma_n = 32$ is a constant value.

## 7.2 Temporal Filtering

Temporal filtering enables us to use stratified sampling as discussed in Chapter 4.2. Filtering is implemented by accumulation of samples via exponentially moving average with temporal sample position projected using a velocity vector:

$$S_i = (1 - \alpha)E_i + \alpha S_{i-1}, \tag{7.3}$$

This is an irradiance approximation based on work by Karis [23] in temporal antialiasing. Unfortunately, using a constant value for $\alpha$ would cause severe ghosting artifacts. Furthermore, due stratified sampling causing extreme variance in the irradiance values between frames, color space clipping methods used in antialiasing are ill-suited for the task. Thus, we choose to rely on geometry based methods and define $\alpha$ as:

$$\alpha = 0.95 \; w_z(p, q) \; w_n(p, q), \tag{7.4}$$

where $p$ is a current pixel sample and $q$ is the projected sample from previous frame. Since the weight function requires temporal surface parameters, we must maintain the normal and depth data from G-buffer of the previous frame. If the resolution of the irradiance estimation target was lower than the filtering target, we upscale the splatting result at the begin of the temporal filtering pass by using bilinear sampling.

## 7.3 Spatial Filtering

Edge-avoiding Á-Trous wavelet transform is a multi-pass algorithm that performs a stationary wavelet transformation iteration for each pass with increasing kernel footprint ($\Omega$). This is illustrated in Figure 7.1. However, we also notice that the number of nonzero elements in the kernel stays the same for all iterations and that correlation within those elements remain. This allows us to ignore the zero elements and keep the size of the computation kernel constant while using group shared memory to efficiently share the surface attributes within the kernel.

Our implementation follows the previous work by Dammertz et al. [15] and Schied et al. [33] in which we realize each wavelet iteration as a 5 x 5 *cross bilateral filter*. Contributing samples are weighted by a weight function

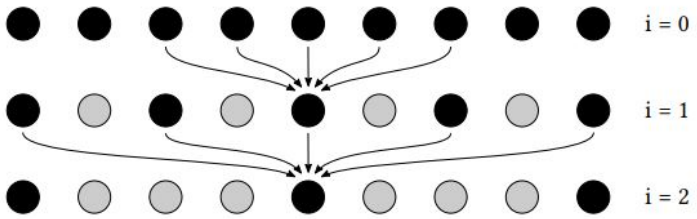



*Figure 7.1: Three iteration of 1D stationary wavelet transform. Arrows showing the nonzero elements of the previous result contribution to current element, while gray dots are zero elements.* (Illustration after Dammertz et al. [15].)

$w(p,q)$, where $p$ being the current pixel and $q$ the contributing sample pixel within the filter. Therefore, we calculate the scaling coefficient $S_{i+1}$ by:

$$S_{i+1} = \frac{\sum_{q\in\Omega} h(q)w(p,q)S_i(q)}{\sum_{q\in\Omega} h(q)w(p,q)}, \tag{7.5}$$

where $h(q) = (\frac{1}{8}, \frac{1}{4}, \frac{1}{2}, \frac{1}{4}, \frac{1}{8})$ is the filter kernel and $w(p,q) = w_z(p,q)\ w_n(p,q)$.

Previous work with edge-avoiding Á-Trous wavelet transform that we reference above only store the scaling coefficient of the wavelet transformation. However, this results in excessive blurring of the irradiance and lose of the illumination detail. Schied et al. [33] prevent this by using variance-guiding as a part of the weight function which works well for high-frequency noise but is unsuitable for the low frequency noise caused by the irradiance estimation. Thus, we represent a new filtering algorithm based on the variance clipping of the detail coefficients.

### 7.3.1 Variance Clipping of the Detail Coefficients

Stationary wavelet transform (SWT) was originally introduced to combat one of the shortcomings of the discrete wavelet transform: transformation not being shift-invariant. This was solved in SWT by saving detail coefficients per pixel for each iteration. The detail coefficients can be calculated as follows:

$$D_i = S_i - S_{i+1}, \tag{7.6}$$

This makes the SWT inherently redundant. However, let us consider how to reconstruct the original signal:

$$S_0 = S_N - \sum_{i=N-1} D_i,$$

where $N$ is the number of iterations. As we can see, to reconstruct the original signal we need only the sum of detail coefficients instead of separated coefficients for each of the iterations. This allows us to reduce the amount of required memory to two textures with resolution of the original image. Nevertheless, this just leaves us in the same point where we started—the original image. However, we can modify each of detail coefficients before we add them into the sum. This is done by using variance clipping [32] to define a color space boundaries (CSB) based of the variance within the spatial kernel. These boundaries can be defined as:

$$\mu = \frac{\mathcal{M}_1}{w_\Omega}$$

$$\sigma = \sqrt{\frac{\mathcal{M}_2}{w_\Omega} - \mu^2},$$

$$CSB = \gamma\sigma eq \tag{7.7}$$

where $w_\Omega$ is the sum of sample weights in the kernel footprint and $\mathcal{M}_1$, $\mathcal{M}_2$ are weighted 1st and 2nd color moments:

$$w_\Omega = \sum_\Omega w_i$$

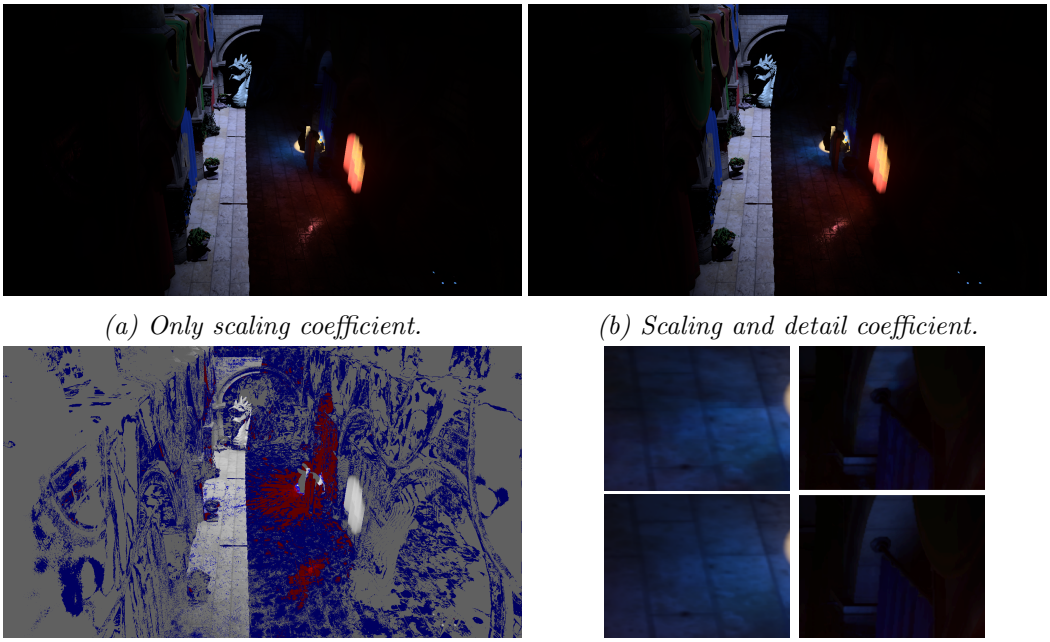
$$\mathcal{M}_1 = \sum_\Omega w_i C_i$$

$$\mathcal{M}_2 = \sum_\Omega w_i C_i^2$$

$\gamma$ is a scaling factor for the variance clipping which is defined as a constant. These boundaries are then used for clipping of the detail coefficients as follows:

$$E_{\text{final}} = S_N - \sum_{i=N-1} \text{clamp}(-CSB_i, CSB_i, D_i), \tag{7.8}$$

The difference in the final illumination result between merely using scaling coefficients and when they are combined with variance clipped detail coefficients is demonstrated in Figure 7.2. It is worth noticing that this clipping is temporally stable only because of the shift-invariant nature of the SWT and would not work for standard discrete wavelet transformation.



*(a) Only scaling coefficient.* *(b) Scaling and detail coefficient.*

*(c) Difference.* *(d)* *(e)*

*Figure 7.2: Comparison between using only scaling coefficients and combination of scaling and variance clipped detail coefficients. As we can see, utilizing detail coefficients can prevent some of the excessive blurring (Figure 7.2d). However, in areas with low photon density this can lead to noise (Figure 7.2e). Therefore, manual tuning of $\gamma$ is often required to achieve the highest quality results.*

## 7.4 Evaluating the Reflected Radiance

Finally, we must apply the filtered irradiance to surface. This results:

$$L_{\text{final}} = E_{\text{final}} \cdot f_{\text{BRDF}}, \tag{7.9}$$

where $L_{\text{final}}$ is the radiance of the global illumination. However, this does not include all necessary directional information.

Photon mapping includes the directional information as an implicit part of the irradiance calculation as surfaces with their geometry normals ($\omega_{gn}$) pointing toward the incoming light direction have a higher probability to be hit by photons. However, this does not capture the detail provided by material attributes, such as normal maps. This is commonly known issue with precalculated global illumination methods and there have been several approaches to solve this [28]. To achieve comparable illumination quality, we must also take this into consideration with photon mapping.

We choose to approach this with a solution inspired by Heitz et al. [17]: We filter the light direction, $\omega_i$, as a separate term and applying irradiance, then we divide by the cosine term between the raw geometrical normal and weighted average of the light direction. Afterwards, we multiply the result by the cosine term between the weighted average light direction ($\omega_i$) and the surface normal including the normal map ($\omega_g$). This changes Equation 7.9 to:

$$L_{\text{final}} = E_{\text{final}} \; f_{\text{BRDF}} \; (\omega_i \cdot \omega_{gn}) \; \min\left(\frac{1}{(\omega_i \cdot \omega_g) + \varepsilon}, s_{lmax}\right) \tag{7.10}$$

This comes with a performance cost as it requires an additional blending target for the irradiance estimate in addition to additional input and output for each filtering step. However, this allows us to apply the information from the normal maps without reading the surface attributes during the irradiance estimation.

Filtering the BRDF, instead of just the light direction, will achieve more accurate results for specular surfaces. However, this would require evaluating the BRDF during the irradiance estimation and thus reading the material attributes. This would come with a significant performance cost when implemented with splatting as the reads of the surface attributes would have to be done for each pixel shader invocation. However, a compute shader that does gathering avoids this problem by loading the material attributes only once.

# Chapter 8

# Results

We assessed the algorithm in three scenes, based on use case, and present our results as follows: First, we show the results of the rendering passes, which is shown only for a single scene as the purpose is to provide an example how these passes contribute to the final result. Second, we show the results and performance numbers for all three test scenes: *Conference room*, *Sponza* and *3DMark Port Royal*. Third, we compare performance in Spoza using different settings configurations to showcase how these parameters contribute to overall performance. Forth, we provide some quality comparison between different rendering settings. Finally, we show modified Sponza scene with transparent and mirror like surfaces to showcase caustics.

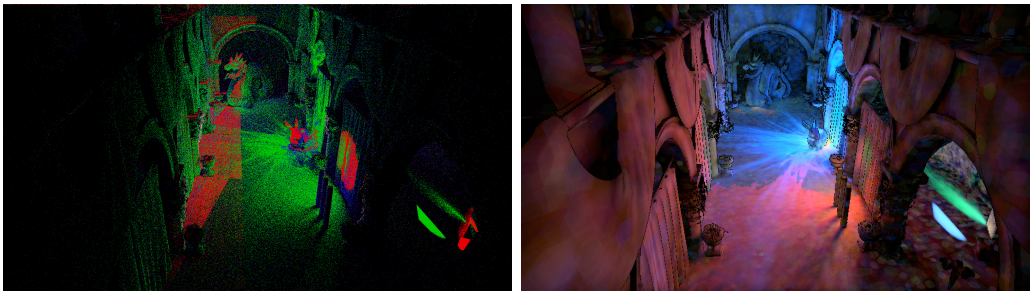
Results for each test scene include, the total illumination result as well as the separated contribution of the local and global illuminations. In addition, they include a reference image rendering using a forward path tracer described in Chapter 3.3.1 using 8192 samples per pixel. However, the implementation of our reference path tracer is quite naive and it is unable to handle correctly some of the test cases, since these test cases were selected photon mapping in mind (e.g., mirror like surfaces). Therefore, it is recommendable not to consider this as a ground true.

Furthermore, we provide a simple description of each test and as well as rendering parameters used for result images. These parameters include i. a. the amount of spatial filtering passes, number of layers allowed for a tile in the screen space irradiance estimation ($n_{tile}$ in Equation 6.2, in practice acts as uniform scaling multiplier to irradiance distribution kernel) and the variance clipping constant for detail coefficients ($\gamma$ in Equation 7.3.1). We also provide computation time of each of the passes: reflective shadow maps (RSM), photon tracing (PT), screen space irradiance estimation (SSIE) and filtering (F).

The results were measured using an NVIDIA GeForce RTX 2080 Ti.

## 8.1 Result for Different Passes

In Figure 8.1 we show the results of various passes of our implementation of the photon mapping algorithm in a modified Sponza scene.

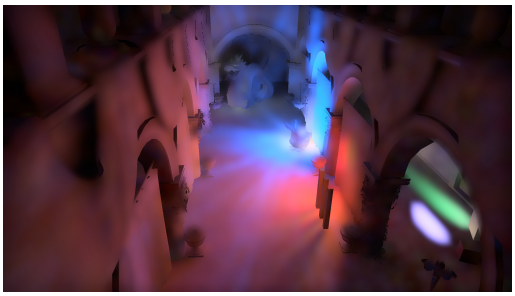


*(a) Photon locations*

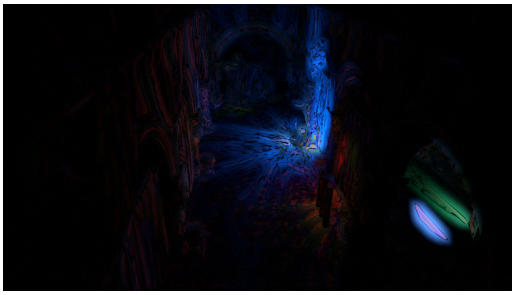


*(b) Splatting result*

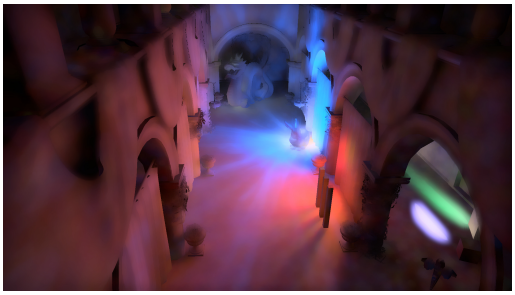
*(c) Temporal filtering result*



*(d) Scaling coefficients*



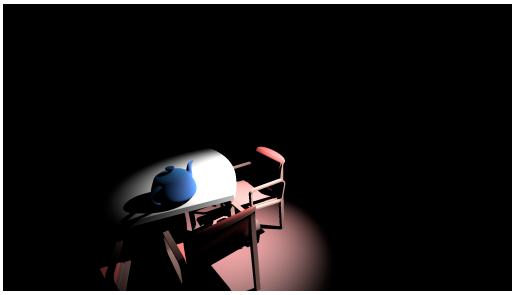
*(e) Variance clipped detail coefficients*



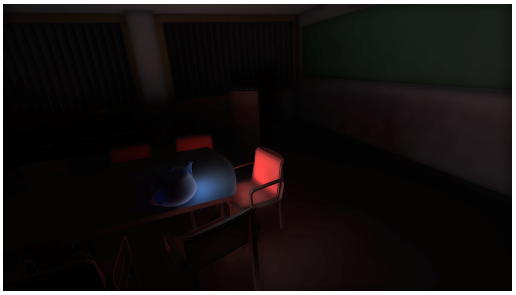
*(f) Irradiance result*

*Figure 8.1: Results for different passes of the algorithm. Sponza scene with three bounces of indirect light, four light sources, three million initial photons and four spatial filtering iterations. We can notice the accumulation of different sample subsets from stratified sampling in temporal filtering result 8.1c compared to the splatting result 8.1b. Also, the effect of the variance clipping of detail coefficients 8.1e is clearly visible as the irradiance result 8.1f retain much of detail that is lost when only scaling coefficients 8.1d are used. In 8.1a the colors of red, green and blue correspond to the number of the bounce.*

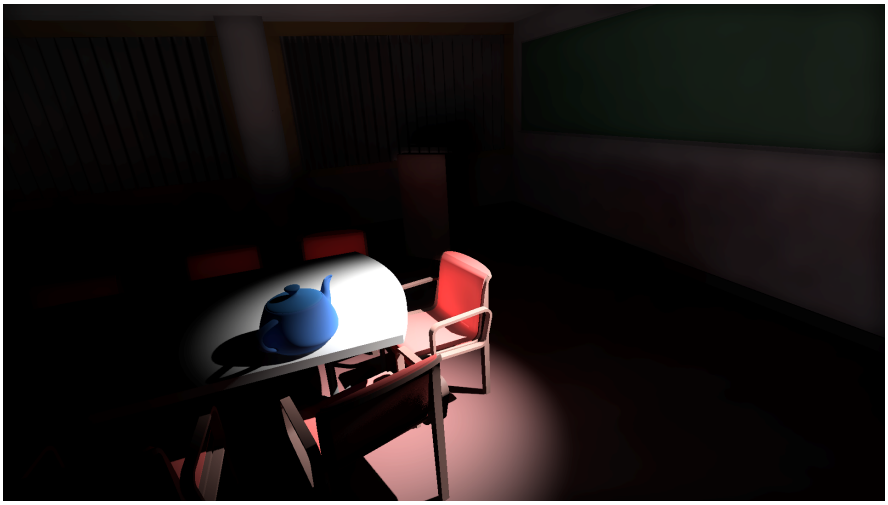
## 8.2 Conference Room



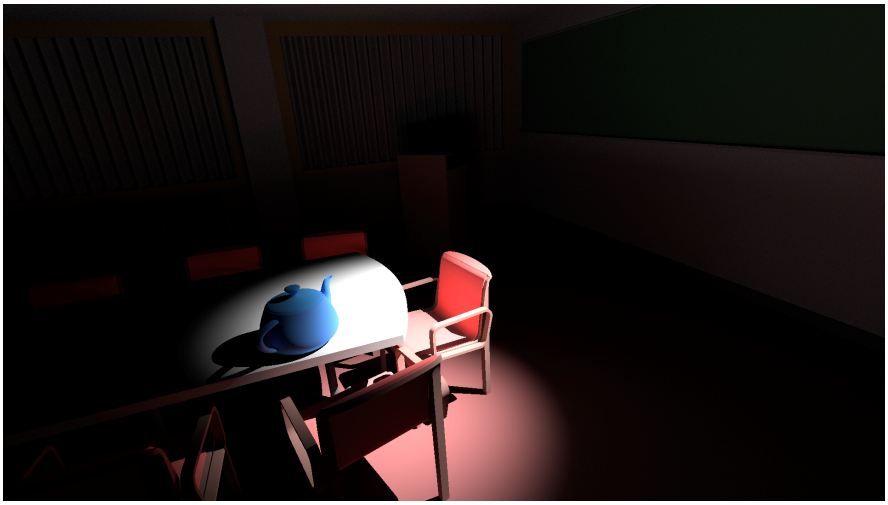
*(a) Direct illumination only.*



*(b) Global illumination only.*



*(c) Final result.*



*(d) Reference.*

*(a)* *(b)*

*Figure 8.3*

Conference room test scene has a single frustum light pointing from the roof to the edge of the table using a single bounce of indirect light. Exposure of the images has been increased to make them more visible in the final print.

As we can see, the overall lighting effects are near correct. However, due to still relatively low sample counts, photon mapping is clearly missing the level of detail present in the reference. Furthermore, there are several artifacts typical of the photon mapping — darkening at the corners of the geometry and the low frequency noise clearly visible at right wall. These problems are only partially addressed by the filtering that can also cause some artifacts, such as banding artifacts (Figure 8.3a) and light bleeding (Figure 8.3b).

| Photons | F-iter | Res. | $n_{tile}$ | $\gamma$ | RSM | PT | SSIE | F | **Total** |
|---|---|---|---|---|---|---|---|---|---|
| 1 M | 4 | 1080p | 1.7 | 0.05 | 0.6 | 0.7 | 4.1 | 3.0 | **8.5** |

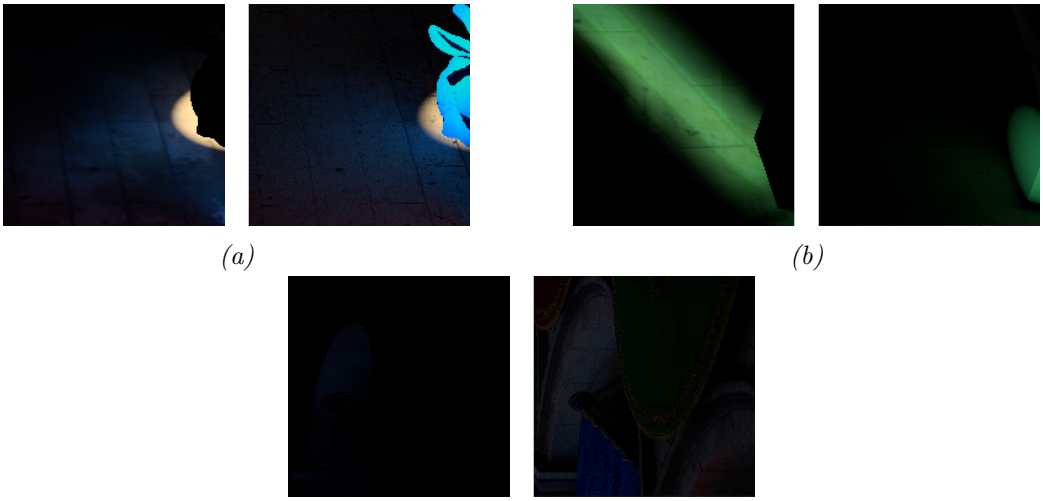
## 8.3 Sponza

*(a) Direct illumination only.*

*(b) Global illumination only.*

*(c) Final result.*

*(d) Reference.*



*(a)* *(b)* *(c)*

*Figure 8.5*

In the modified Sponza test scene contains are three spot lights: One pointing to a Stanford bunny model with blueish mirror-like material, another pointing to the red curtain and the final one directed to a mirror cube on the bottom-right corner. There's also a fourth frustum light representing incoming moonlight. It is worth noting that the intensity of this light source is significantly smaller compared to the spot lights and therefore only a modest amount of photons allocated for it.

As the result of lack of photons, the sampling rate for the moon light is slow and combined with the relatively large volume of the lighting effect, it hardly contributes to the photon mapping's illumination result (Figure 8.5c). This partially compensated by adding *reflections* to the illumination (Figure 8.6). Reflection is the specular component of the global illumination in real-time applications generally handled separately from the main global illumination, but which is included to the path traced result. However, this does not remove the limitation to the photon mapping that much more fundamental: Compared to e.g. path tracing, photon mapping is better suited for local effects, whereas larger ones, such as sun's illumination of open space, would require enormous amount of photons to achieve high-quality illumination. Fortunately, these uses cases are handled relativity well by static global illumination methods.

In the cases of more local spot lights, and especially with mirror-like surfaces, photon mapping actually handles itself better than our reference: In Figure 8.5a we can see it maintaining the illumination effect caused by the

*Figure 8.6: Final rendering result: This is the final rendering results from the 3dMark engine, which include in addition the reflection i.a. post processing effects. However, it still provides some perception how much reflection contribute the final images.*

shape of the bounced surface, whereas path tracing even with high sample counts just blurs the results. Furthermore, in Figure 8.5b path tracer fails completely to accumulate the reflected shape of the light for the mirror cube.

| Photons | F-iter | Res. | $n_{tile}$ | $\gamma$ | RSM | PT | SSIE | F | **Total** |
|---|---|---|---|---|---|---|---|---|---|
| 1 M | 4 | 1080p | 1.25 | 0.04 | 1.8 | 1.6 | 8.0 | 3.6 | **15.0** |

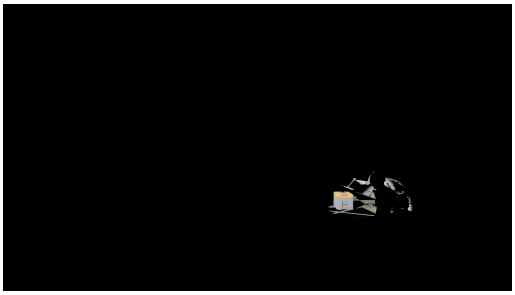
It is worth noting that even with the same amount of photons, the cost of screen space irradiance has increased compared to "Conference room"-test scene. This is mostly caused by the concentration of photons to certain screen space areas, which despite the downscaling of the distribution kernel causes a bottleneck in the pixel blending. This causes the cost of irradiance estimation to be unpredictable and it is something we consider an area of future work. Furthermore, we see the increase of time spent for reflective shadow maps, which is to be expected as it is partially correlated to the amount of light sources.

This scene, albeit from a different angle, highlights another issue related to the optimization of averaging the incoming light direction described in Chapter 7.4. Artifact is not very notable in images, but temporally it much

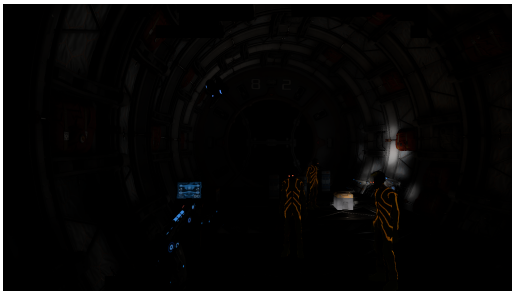
more noticeable as it causes the specular highlight to fluctuate between frames. This widely known issue e.g. in relation to *directional light maps* that use similar approximation and solving this is also considered future work. This artifact is showcased in Figure 8.7.

*Figure 8.7: Artifact caused by averaging of incoming light directions.*

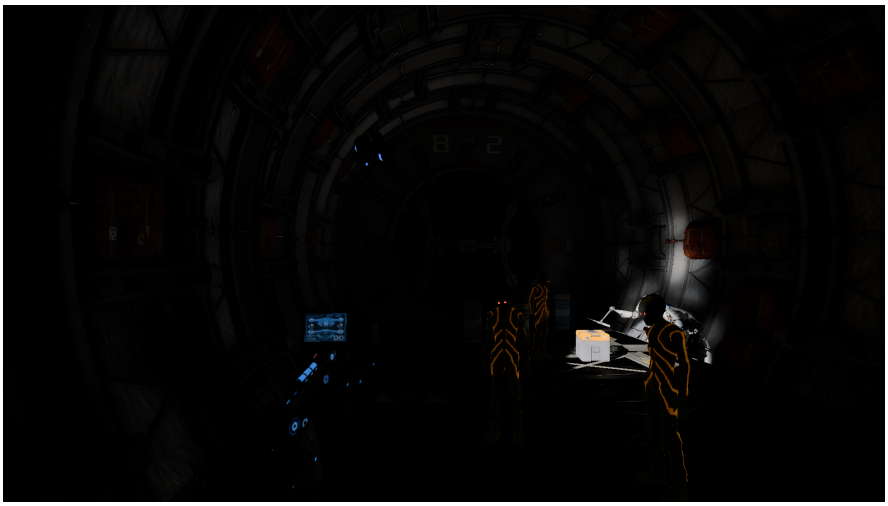
## 8.4 3DMark Port Royal



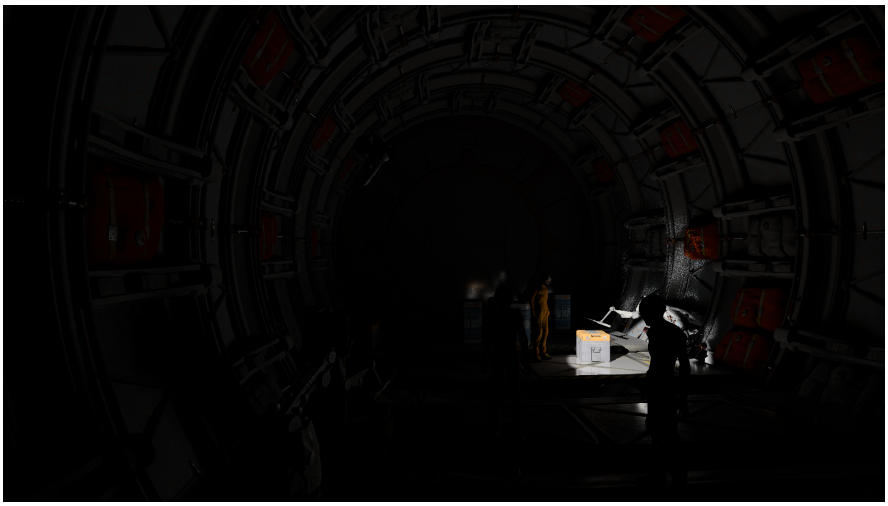
*(a) Direct illumination only.*



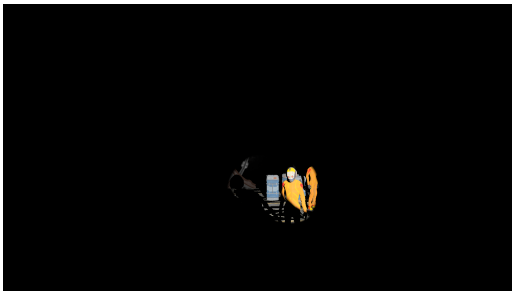
*(b) Global illumination only.*



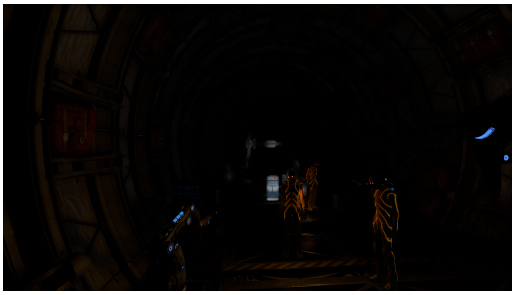
*(c) Final result.*



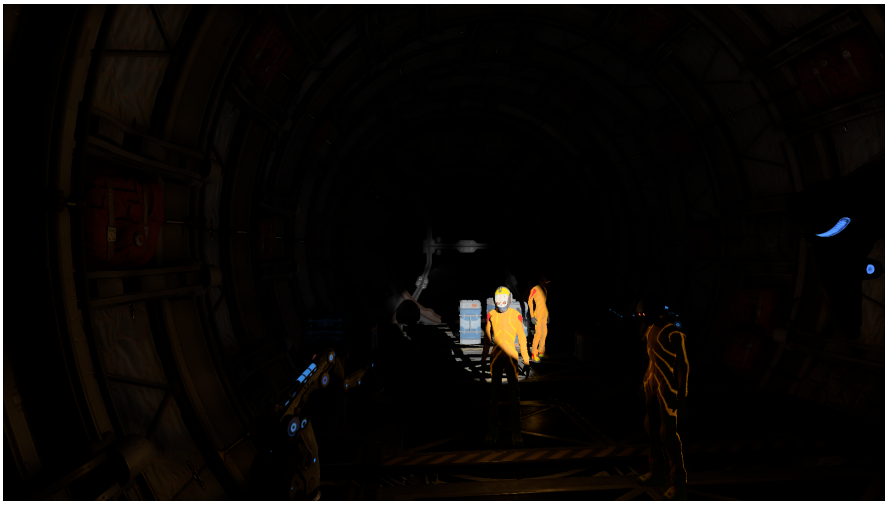
*(d) Reference.*



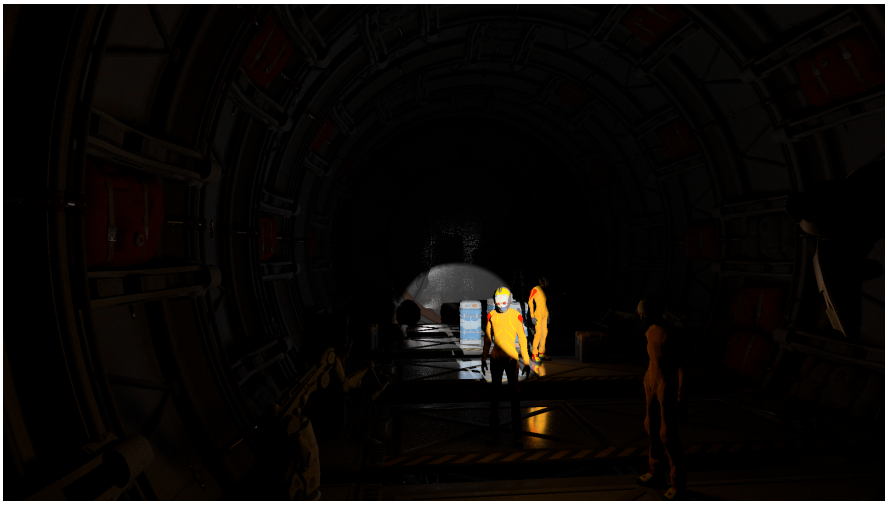
*(a) Direct illumination only.*



*(b) Global illumination only.*



*(c) Final result.*



*(d) Reference.*

Two snapshots from the 3dMark Port Royal, which has a single spot light located at the head of the flying drone.

| Photons | F-iter | Res. | $n_{tile}$ | $\gamma$ | RSM | PT | SSIE | F | **Total** |
|---|---|---|---|---|---|---|---|---|---|
| 1 M | 4 | 1080p | 1.25 | 0.04 | 0.6 | 0.9 | 1.8 | 3.6 | **7.0** |
| 1 M | 4 | 1080p | 1.25 | 0.04 | 0.7 | 1.2 | 1.1 | 3.0 | **6.1** |

## 8.5 Further Performance Measurements

Table 8.1 reports the computation times as milliseconds for the test scenes described above while using high quality settings — 3 million photons and three bounces of indirect lighting. This was done to provide examples of the scalability compared to results in previous passes. Furthermore, in Table 8.2 we examine the effect of different rendering settings to the individual passes of the algorithm.

| *Scene* | RSM | Tracing | Splatting | Filtering | **Total** |
|---|---|---|---|---|---|
| *Conference room* | 1.6 | 5.2 | 7.5 | 3.3 | **17.6** |
| *Sponza* | 2.1 | 3.0 | 5.5 | 3.6 | **14.2** |
| *3DMark Port Royal* | 2.1 | 8.0 | 8.3 | 3.3 | **21.7** |

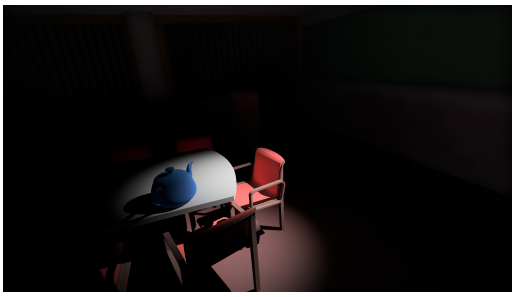
*Table 8.1: Performance of the photon mapping algorithm per scene. These performance numbers were measured using extreme high settings to increase and can be considered the highest end settings for the photon mapping.*

| Photons | Bounce | Reso. | RSM | Tracing | Splatting | Filtering | **Total** |
|---|---|---|---|---|---|---|---|
| 1.0 mm | 1 | 1080p | 1.4 | 0.7 | 1.2 | 3.1 | **6.1** |
| 1.0 mm | 1 | 1440p | 1.4 | 0.7 | 1.6 | 5.6 | **9.3** |
| 2.0 mm | 1 | 1080p | 1.8 | 1.3 | 2.3 | 3.1 | **9.0** |
| 3.0 mm | 1 | 1080p | 2.1 | 1.8 | 3.9 | 3.1 | **10.8** |
| 1 mm | 3 | 1080p | 1.4 | 1.3 | 2.1 | 3.1 | **7.9** |

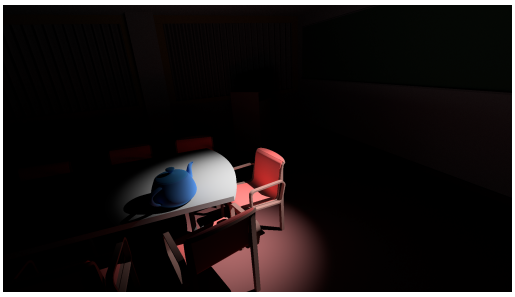
*Table 8.2: Performance of the photon mapping algorithm using different settings in the Sponza scene, but from different camera and moment in timeline compared to previous results. Filtering is done with four spatial iterations. The baseline is set to what we would consider "low" settings for photon mapping — 1 million photons and a single bounce.*

## 8.6 Quality Comparison with Different Rendering Settings

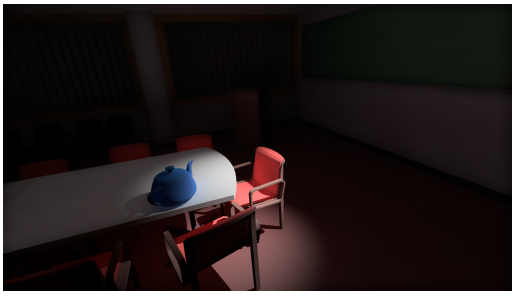
### 8.6.1 Number of Indirect Bounces



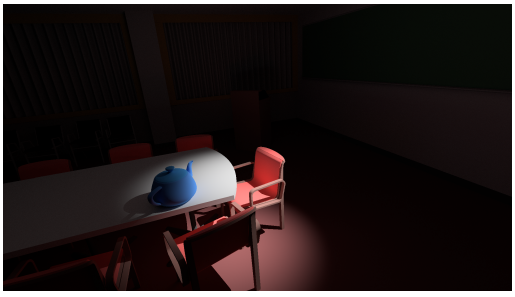
*(a) Photon mapping, a single bounce.*



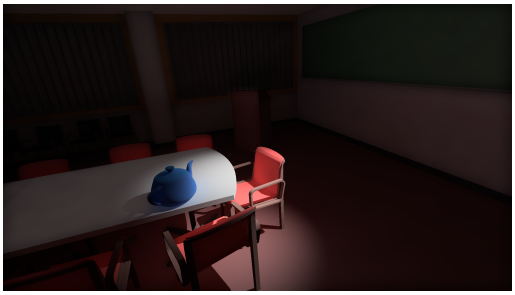
*(b) Reference, a single bounce.*



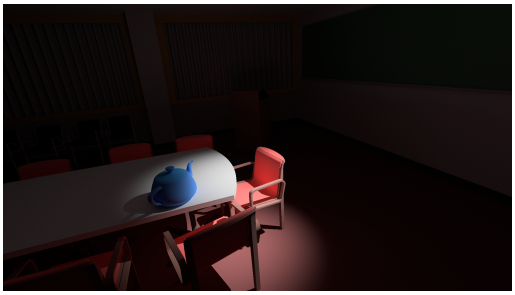
*(c) Photon mapping, two bounces.*



*(d) Reference, two bounces*



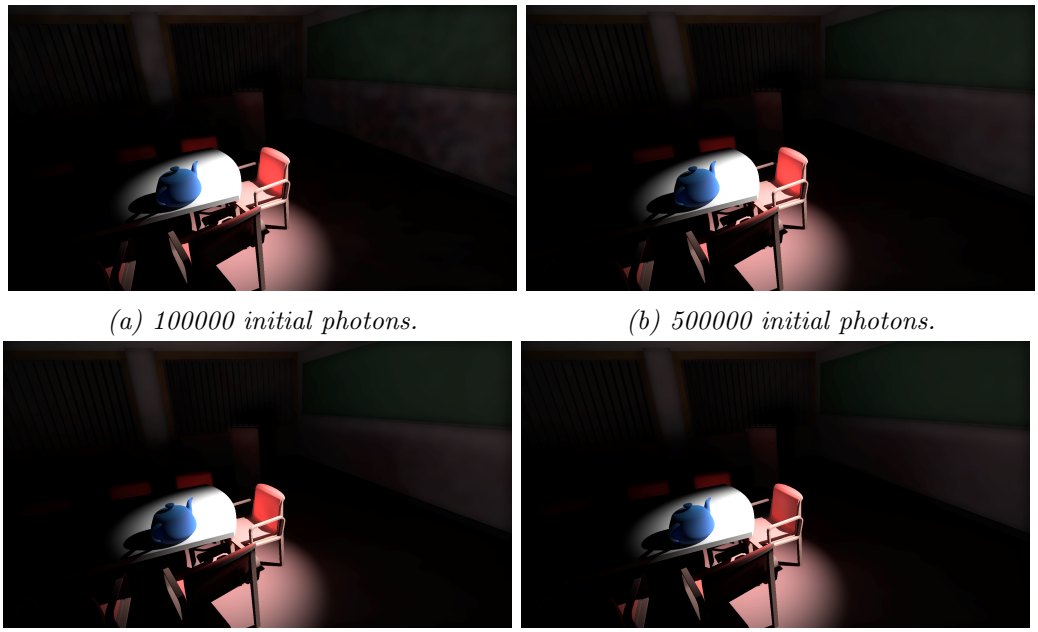
*(e) Photon mapping, three bounces.*



*(f) Reference, three bounces.*

| Photons | F-iter | Boun. | $n_{tile}$ | $\gamma$ | RSM | PT | SSIE | F | **Total** |
|---|---|---|---|---|---|---|---|---|---|
| 1 M | 4 | 1 | 1.7 | 0.05 | 0.8 | 0.5 | 4.1 | 3.0 | **8.4** |
| 1 M | 4 | 2 | 1.7 | 0.05 | 1.0 | 1.3 | 12.4 | 3.4 | **18.4** |
| 1 M | 4 | 3 | 1.7 | 0.05 | 0.8 | 1.6 | 17.6 | 3.6 | **23.6** |

### 8.6.2 Number of Initial Photons



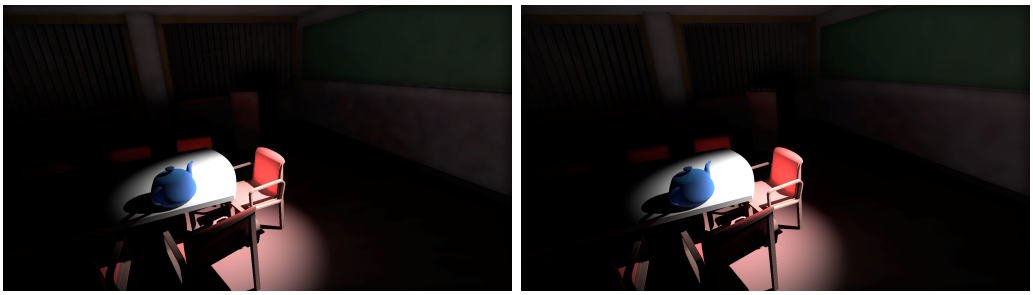
*(a) 100000 initial photons.* *(b) 500000 initial photons.*

*(c) 1M initial photons.* *(d) 3M initial photons.*

*Figure 8.11: Comparison of different initial photon counts. Exposure of the images has been increased.*

| Photons | F-iter | Boun. | $n_{tile}$ | $\gamma$ | RSM | PT | SSIE | F | **Total** |
|---|---|---|---|---|---|---|---|---|---|
| 0.1 M | 4 | 1 | 1.7 | 0.05 | 0.5 | 0.1 | 0.4 | 3.0 | **4.1** |
| 0.5 M | 4 | 1 | 1.7 | 0.05 | 0.6 | 0.3 | 2.6 | 3.4 | **7.0** |
| 1 M | 4 | 1 | 1.7 | 0.05 | 0.6 | 0.7 | 4.1 | 3.0 | **8.5** |
| 3 M | 4 | 1 | 1.7 | 0.05 | 1.3 | 1.7 | 13.5 | 3.7 | **20.2** |

### 8.6.3 Number of Spatial Filter Iterations

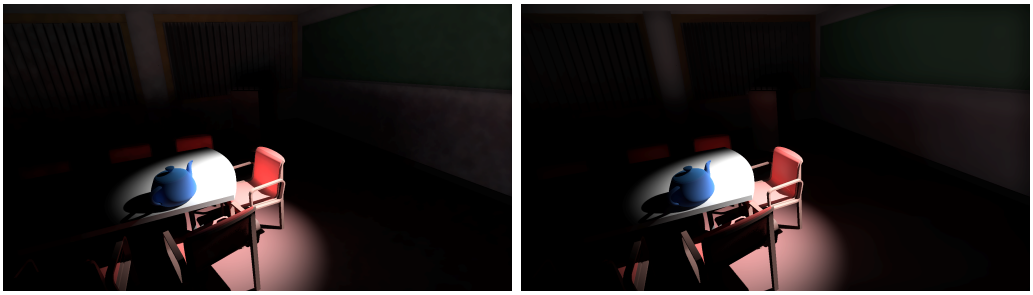


*(a) One iteration of the spatial filter.* *(b) Four iterations of the spatial filter.*

*Figure 8.12: Comparison between one and four spatial filter iterations. Exposure of the images has been increased.*

| Photons | F-iter | Boun. | $n_{tile}$ | $\gamma$ | RSM | PT | SSIE | F | **Total** |
|---|---|---|---|---|---|---|---|---|---|
| 0.5 M | 1 | 1 | 1.7 | 0.05 | 0.7 | 0.5 | 2.4 | 1.4 | **5.1** |
| 0.5 M | 4 | 1 | 1.7 | 0.05 | 0.6 | 0.3 | 2.6 | 3.4 | **7.0** |

### 8.6.4 Number of Photons per Density Tile — $n_{tile}$



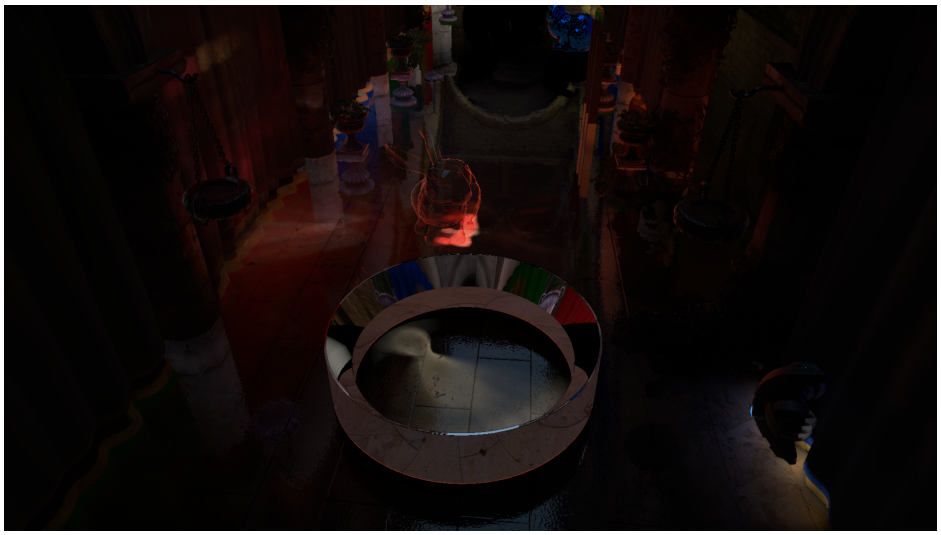
*(a)* $n_{tile} = 0.5$ *(b)* $n_{tile} = 3.0$

| Photons | F-iter | Boun. | $n_{tile}$ | $\gamma$ | RSM | PT | SSIE | F | **Total** |
|---|---|---|---|---|---|---|---|---|---|
| 0.5 M | 4 | 1 | 0.5 | 0.05 | 0.7 | 0.5 | 0.9 | 3.0 | **5.2** |
| 0.5 M | 4 | 1 | 3.0 | 0.05 | 0.7 | 0.5 | 10.6 | 3.4 | **15.2** |

## 8.7 Showcase of Caustics

This modified Sponza scene has two spot lights: The first of these spotlights points to the transparent Stanford bunny from above. The second light is pointing towards the mirror-like torus from the flying drone at the front.

*(a) Global illumination only*



*(b) Final result*

# Chapter 9

# Conclusion

In this thesis, we have presented an introduction to computing global illumination using Monte Carlo rendering methods and then expanding one of those methods, photon mapping, to be available for real-time applications through several optimization methods: We expand the concept of reflective shadow maps related to the sampling of local illumination for photon mapping by utilizing importance sampling and Markov Chains. We also introduce an approach relying on stratified sampling to temporally increase photon map's sampling rate by accumulating samples over multiple frames. Finally, we expanded the definition of common Russian Roulette optimization for photon reflections based on microfacet BRDF model.

For screen space estimation, we introduce a new anisotropic distribution kernel, which accounts for e.g. ray differentials and the screen space density estimation of the photon map. Furthermore, we provide comparison for two of the most prevalent methods to generate a screen space irradiance estimate.

We also introduced a new filtering method designed for low frequency noise present in the irradiance estimation by utilizing color space variance clipping for detail coefficients of the Á-Trous wavelet transform and using the clipped coefficient for the re-creation process of the image.

This research was motivated by the goal to increase dynamism of the current cutting-edge real-time global illumination solutions, which are mostly dependent on precomputed illumination data. From this point of view, we have been successful: Our algorithm can provide real-time global illumination results accumulated over a few frames for a limited set of light sources. Furthermore, our algorithm does not rely on the existence of any global illumination methods or their related parameterizations, such as unique sampling coordinates for light maps or probe placement for illumination probes. Therefore, our algorithm can be applied with relative ease to an existing rendering pipeline.

As mentioned in Chapter 8.3, photon mapping is better suited for more local illumination effects and would require excessive sample counts in more global effects, such as open spaces. Fortunately, these global use cases are quite well handled by static methods. Furthermore, our algorithm does not exclude any of the recently introduced methods [7] [24], which add dynamism by updating the precomputed data structures in real-time.

The aforementioned global effects also bring up an interesting question regarding which of the rendering passes is going to be the algorithm's bottleneck in the future: Currently this can vary as with more localized cases the irradiance estimate becomes the bottleneck ( 8.6.2) while the with the global ones it is the photon tracing. This could possible mean that even more global effects could be possibly for real-time photon mapping as the GPU ray tracing performance keeps increasing but it is questionable whether it is more efficient than e.g. path tracing.

However despite our efforts, sample counts available for real-time time applications are not always enough to achieve desired illumination and there remains a trade-off between rendering quality, performance and artifacts caused by different approximations. Not to mention that our algorithm is probably too costly in terms of performance to be currently used for e.g. video games. Therefore, there is a necessity for future work.

## 9.1 Future Work

**Optimizing Screen Space Irradiance Estimation** As discussed in result section, splatting of the photons can easily become bottleneck for the algorithm and maybe even worse, its performance can be very unpredictable. In the cases of high-density photon density, the screen-space size of the distribution kernel can even approach the size of a single pixel, which makes drawing the splatting kernel wasteful. This could possibly be solved by writing the irradiance value directly to the framebuffer instead of splatting, but this has some technical challenges due limited availability of atomic operations. Furthermore, it could be possible to optimize the splatting using a new GPU-feature called *variable rate shading*. Finally, there is still possibility that with some improvements to the memory coherency and better culling, tile-based scattering might prove to be a better approach.

**Adaptive Constants for Variance Clipping of the Detail Coefficients** Unfortunately, we cannot determine if the variance in the irradiance is caused by the low sample count or an actual difference in lighting conditions. This is partly mitigated by the larger sample set provided by stratified sampling. As

these samples are accumulated using temporal filtering, the noise becomes visible in cases where temporal samples are being rejected. Therefore, it would be preferable to use less constricting variance clipping boundaries for these areas. Such a system could be implemented by scaling the variance clipping constant based on the weights that we use to define the accumulation of the temporal samples.

**Improving the Directional Component of Illumination** Our current approach of averaging the incoming light directions works fairly well as approximation but it can cause some notable artifacts, especially with with specular surfaces. There is several possible approaches to solve this, such as averaging the entire BRDF [40] or using spherical harmonics as was done in Nvidia's Quake II RTX [2].

# Bibliography


[1] Compute shader overview - windows applications — microsoft docs. https://docs.microsoft.com/en-us/windows/desktop/direct3d11/direct3d-11-advanced-stages-compute-shader. (Accessed on 06/23/2019).

[2] Github-nvidia/q2rtx: Nvidia implementation of rtx ray-tracing in quake2. https://github.com/NVIDIA/Q2RTX. (Accessed on 06/29/2019).

[3] Id3d12graphicscommandlist::drawinstanced (d3d12.h) — microsoft docs. https://docs.microsoft.com/en-us/windows/desktop/api/d3d12/nf-d3d12-id3d12graphicscommandlist-drawinstanced. (Accessed on 06/23/2019).

[4] Indirect drawing and gpu culling - windows applications — microsoft docs. https://docs.microsoft.com/en-us/windows/desktop/direct3d12/indirect-drawing-and-gpu-culling-. (Accessed on 06/23/2019).

[5] Weak law of large numbers – from wolfram mathworld. http://mathworld.wolfram.com/WeakLawofLargeNumbers.html. (Accessed on 06/23/2019).

[6] Akenine-Moller, T., Haines, E., and Hoffman, N. *Real-time rendering*. AK Peters/CRC Press, 2018.

[7] Apers, D., Edblom, P., de Rousiers, C., and Hillaire, S. Interactive light map and irradiance volume preview in frostbite. In *Ray Tracing Gems*. Springer, 2019, pp. 377–407.

[8] Barák, T., Bittner, J., and Havran, V. Temporally coherent adaptive sampling for imperfect shadow maps. In *Computer Graphics Forum* (2013), vol. 32, Wiley Online Library, pp. 87–96.

[9] Buehler, C., Bosse, M., McMillan, L., Gortler, S., and Cohen, M. Unstructured lumigraph rendering. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM, pp. 425–432.

[10] Christensen, P. H., Jarosz, W., et al. The path to path-traced movies. *Foundations and Trends® in Computer Graphics and Vision 10*, 2 (2016), 103–175.

[11] Clarberg, P., Jarosz, W., Akenine-Möller, T., and Jensen, H. W. Wavelet Importance Sampling: Efficiently Evaluating Products of Complex Functions. *ACM Transactions on Graphics 24*, 3 (2005), 1166–1175.

[12] Cook, R. L., and Torrance, K. E. A reflectance model for computer graphics. In *ACM Siggraph Computer Graphics* (1981), vol. 15, ACM, pp. 307–316.

[13] Dachsbacher, C., and Stamminger, M. Reflective Shadow Maps. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games* (2005), pp. 203–231.

[14] Dachsbacher, C., and Stamminger, M. Splatting Indirect Illumination. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games* (2006), ACM, pp. 93–100.

[15] Dammertz, H., Sewtz, D., Hanika, J., and Lensch, H. Edge-Avoiding À-Trous Wavelet Transform for Fast Global Illumination Filtering. In *Proceedings of High-Performance Graphics* (2010), pp. 67–75.

[16] Heitz, E., and d'Eon, E. Importance Sampling Microfacet-Based BSDFs using the Distribution of Visible Normals. *Computer Graphics Forum 33*, 4 (2014), 103–112.

[17] Heitz, E., Hill, S., and McGuire, M. Combining Analytic Direct Illumination and Stochastic Shadows. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2018), pp. 2:1–2:11.

[18] Hoffman, N. Background: physics and math of shading. *Physically Based Shading in Theory and Practice 24*, 3 (2013), 211–223.

[19] Igehy, H. Tracing ray differentials. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (1999), ACM Press/Addison-Wesley Publishing Co., pp. 179–186.

[20] Jaquays, P., and Hook, B. Quake iii arena.

[21] Jensen, H. W. *Realistic Image Synthesis Using Photon Mapping*. A K Peters, 2001.

[22] Kalos, M. H., and Whitlock, P. A. *Monte carlo methods*. John Wiley & Sons, 2009.

[23] Karis, B. High-Quality Temporal Supersampling. Advances in Real-Time Rendering in Games, SIGGRAPH Courses, 2014.

[24] Majercik, Z., Guertin, J.-P., Nowrouzezahrai, D., and McGuire, M. Dynamic diffuse global illumination with ray-traced irradiance fields. *Journal of Computer Graphics Techniques Vol 8*, 2 (2019).

[25] Mara, M., Luebke, D., and McGuire, M. Toward Practical Real-Time Photon Mapping: Efficient GPU Density Estimation. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2013), pp. 71–78.

[26] McGuire, M., and Luebke, D. Hardware-Accelerated Global Illumination by Image Space Photon Mapping. In *Proceedings of High-Performance Graphics* (2009), pp. 77–89.

[27] McGuire, M., Mara, M., Nowrouzezahrai, D., and Luebke, D. Real-time global illumination using precomputed light field probes. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2017), ACM, p. 2.

[28] O'Donnell, Y. Precomputed Global Illumination in Frostbite. Game Developers Conference, 2018.

[29] of Standards, U. S. N. B., and Nicodemus, F. E. *Geometrical considerations and nomenclature for reflectance*, vol. 160. US Department of Commerce, National Bureau of Standards, 1977.

[30] Oren, M., and Nayar, S. K. Generalization of lambert's reflectance model. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (1994), ACM, pp. 239–246.

[31] Pharr, M., Jakob, W., and Humphreys, G. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[32] Salvi, M. An Excursion in Temporal Supersampling. From the Lab Bench: Real-Time Rendering Advances from NVIDIA Research, Game Developers Conference, 2016.

[33] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination. In *Proceedings of High-Performance Graphics* (2017), pp. 2:1–2:12.

[34] Schlick, C. An inexpensive brdf model for physically-based rendering. In *Computer graphics forum* (1994), vol. 13, Wiley Online Library, pp. 233–246.

[35] Schregle, R. Bias Compensation for Photon Maps. *Computer Graphics Forum 22*, 4 (2003), 729–742.

[36] Shirley, P., Laine, S., Hart, D., Pharr, M., Clarberg, P., Haines, E., Raab, M., and Cline, D. Sampling transformations zoo. In *Ray Tracing Gems*. Springer, 2019, pp. 223–246.

[37] Silvennoinen, A., and Lehtinen, J. Real-time global illumination by precomputed local reconstruction from sparse radiance probes. *ACM Transactions on Graphics (TOG) 36*, 6 (2017), 230.

[38] Smal, N., and Aizenshtein, M. Real-time global illumination with photon mapping. In *Ray Tracing Gems*. Springer, 2019, pp. 409–436.

[39] Smith, B. Geometrical shadowing of a random rough surface. *IEEE transactions on antennas and propagation 15*, 5 (1967), 668–671.

[40] Stachowiak, T., and Uludag, Y. Stochastic screen-space reflections. *ACM SIGGRAPH Courses 2015: Advances in Real-Time Renderingin Games* (2015).

[41] Veach, E., and Guibas, L. J. Optimally combining sampling techniques for monte carlo rendering. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (1995), ACM, pp. 419–428.

[42] Veach, E., and Guibas, L. J. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), ACM Press/Addison-Wesley Publishing Co., pp. 65–76.

[43] Walter, B., Marschner, S. R., Li, H., and Torrance, K. E. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics conference on Rendering Techniques* (2007), Eurographics Association, pp. 195–206.

# Appendix A

# Variance for Monte Carlo Integrator

$$
\begin{aligned}
\mathrm{Var_N} &= E\left[\frac{1}{N-1}\left(\frac{1}{N}\sum_{i=1}^{N}\left(\frac{f(X_i)}{p(X_i)}\right)^2 - \left(\frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}\right)^2\right)\right] \\
&= \frac{1}{N-1}\left(E\left[\left(\frac{f(X)}{p(X)}\right)^2\right] - E\left[\frac{f(X)}{p(X)}\right]^2\right) \\
&= \frac{\mathrm{Var_1}}{N-1} \\
&= O\left(\frac{1}{N}\right)
\end{aligned}
$$

# Appendix B

# Code Sample for Tracing

```
1
2  struct Payload
3  {
4      // Next ray direction and state of random generator
5      uint4 direction_random;
6      // Current photon power, ray length and bounce count
7      uint3 power_t_bounce;
8  };
9
10 [shader("raygeneration")]
11 void rayGen()
12 {
13     Payload p;
14     RayDesc ray;
15
16     // First we either read to initial sample from RSM or use RT to cast it
17     // from the light.
18     #if RAY_TRACE_FROM_LIGHT
19     RayTraceFirstBounce(p);
20     #else
21     ReadRSMSamplePosition(p);
22     #endif
23     // We check if continues bounces by the bounce count
24     // and ray length (zero for terminated trace or miss)
25     while(p.power_t_bounce.z < MAX_BOUNCE_COUNT
26         && p.power_t_bounce.y != 0)
27     {
28         // We get the ray origin and direction for the state
29         float3 dir_in_world = .0f;
30         dir_in_world.xy = from_f16f16(p.direction_random.x);
31         dir_in_world.z = from_f16f16(p.direction_random.y).x;
32         ray.Origin = get_hit_position_in_world(p, ray);
33         ray.Direction = dir_in_world;
34
35         TraceRay(gRtScene, RAY_FLAG_FORCE_OPAQUE, 0xFF, 0, 1, 0, ray, p);
36         p.power_t_bounce.z =  p.power_t_bounce.z + 1;
37     }
38 }
39
40 void validate_and_add_photon(Surface_attributes surface, float3
       position_in_world, float3 power,
41 float3 incoming_direction, float t)
```

```
42  {
43      if(is_in_camera_frustum(position)
44          && is_normal_direction_to_camera(surface.normal))
45      {
46          uint tile_index = get_tile_index_in_flattened_buffer(
                position_in_world);
47          uint photon_index;
48          // Offset in the photon buffer and the indirect argument
49          DrawArgumentBuffer.InterlockedAdd(4, 1, photon_index);
50          // Photon is packed and stored with correct offset
51          add_photon_to_buffer(position_in_world, power, surface.normal, power
                ,
52          incoming_direction, photon_index, t);
53          // Tile based photon density estimation
54          DensityEstimationBuffer.InterlockedAdd(tile_i * 4, 1);
55      }
56  }
57
58  [shader("closesthit")]
59  void closestHitShader(inout Payload p : SV_RayPayload, in
        IntersectionAttributes attribs : SV_IntersectionAttributes)
60  {
61      // Load surface attributes for the hit
62      Surface_attributes surface = LoadSurface(attribs);
63
64      float3 ray_direction = WorldRayDirection();
65      float3 hit_pos = WorldRayOrigin() + ray_direction * t;
66      float3 incoming_power = from_rbge5999(p.power_t_bounce.x);
67      float3 outgoing_power = .0f;
68
69      RandomStruct r;
70      r.seed = p.direction_random.z;
71      r.key = p.direction_random.w;
72
73      // Russian Roulette check
74      float3 out_going_direction = .0f;
75      float3 store_power = .0f;
76      bool keep_going = russian_roulette(incoming_power,  ray_direction,
77      surface, r,  outgoing_power, out_going_direction, store_power);
78
79      repack_the_state_to_payload(r.key, outgoing_power,
80      out_going_direction, keep_going);
81
82      validate_and_add_photon(surface, hit_pos, store_power, ray_direction, t)
            ;
83  }
```

# Appendix C

# Code Sample for Splatting

```
1
2  float uniform_scaling(float3 pp_in_view, float ray_length)
3  {
4      // Tile based culling as photon density estimation
5      float layers =
6          load_number_of_photons_in_tile(pp_in_view);
7      float r = .1f;
8
9      if(layers > .0f)
10     {
11         float d = pp_in_view.z;
12         float A_t = d * d * TileAreaConstant;
13         r = sqrt(A_t / (PI * layers));
14     }
15     r = clamp(r, DYNAMIC_KERNEL_SCALE_MIN,
16         DYNAMIC_KERNEL_SCALE_MAX);
17
18     r = max(.1f,
19             lerp(.0f, 1.0f, ray_length / MAX_RAY_LENGTH));
20     return r * LayerScale;
21 }
22
23 kernel_output kernel_modification_for_vertex_position(
24     float3 vertex, float3 n, float3 light, float3 pp_in_view, float
           ray_length)
25 {
26     kernel_output o;
27     float scaling_uniform =  uniform_scaling(pp_in_view,  ray_length);
28
29     float3 l = normalize(light);
30     float3 cos_alpha = dot(n, vertex);
31     float3 projected_v_to_N = cos_alpha * n;
32     float cos_theta = saturate(dot(n, l));
33     float3 projected_L_to_N = cos_theta * n;
34
35     float3 T = normalize(l - projected_L_to_N);
36
37     o.light_shaping_scale = min(1.0f/cos_theta,
38         EVA_MAX_SCALING_CONSTANT);
39
40     float3 projected_v_to_T = dot(T, vertex) *  T;
41     float3 projected_v_to_U = vertex - projected_v_to_T - projected_v_to_N;
```

```
42
43      float3 scaled_U = projected_v_to_T *  light_shaping_scale *
            scaling_uniform;
44      float3 scaled_T = projected_v_to_U * scaling_uniform;
45      o.vertex_position = scaled_U  + scaled_T +
46      (KernelCompress * projected_v_to_N);
47
48      o.ellipse_area = PI * o.scaling_uniform  * o.scaling_uniform *  o.
            light_shaping_scale;
49
50      return o;
51  }
52
53  void VS(
54  float3 Position : SV_Position,
55  uint instanceID : SV_InstanceID,
56  out vs_to_ps Output)
57  {
58      unpacked_photon up = unpack_photon(PhotonBuffer[instanceID]);
59      float3 photon_position = up.position;
60      float3 photon_position_in_view = mul(WorldToViewMatrix,
61      float4(photon_position , 1)).xyz;
62      kernel_output o = kernel_modification_for_vertex_position(Position,
63      up.normal, -up.direction, photon_position_in_view, up.ray_length);
64
65      float3 p = pp + o.vertex_position;
66
67      Output.Position = mul(WorldToViewClipMatrix, float4(p, 1));
68      Output.Power = up.power / o.ellipse_area;
69      Output.Direction = -up.direction;
70  }
71
72  [earlydepthstencil]
73  void PS(
74  vs_to_ps Input,
75  out float4 OutputColorXYZAndDirectionX : SV_Target,
76  out float2 OutputDirectionYZ : SV_Target1)
77  {
78      float depth = DepthTexture[Input.Position.xy];
79      float gbuffer_linear_depth = LinearDepth(ViewConstants, depth);
80      float kernel_linear_depth = LinearDepth(ViewConstants, Input.Position.z)
            ;
81      float d = abs(gbuffer_linear_depth - kernel_linear_depth);
82
83      clip(d > (KernelCompress * MAX_DEPTH) ? -1 : 1);
84
85      float3 power = Input.Power;
86      float total_power = dot(power.xyz, float3(1.0f, 1.0f, 1.0f));
87      float3 weighted_direction = total_power * Input.Direction;
88
89      OutputColorXYZAndDirectionX = float4(power, weighted_direction.x);
90      OutputDirectionYZ = weighted_direction.yz;
91  }
```